

**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ ВИЩИЙ НАВЧАЛЬНИЙ ЗАКЛАД
ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
КАФЕДРА ПРИКЛАДНОЇ МАТЕМАТИКИ ТА ІНФОРМАТИКИ**

**СХЕМОКОНСПЕКТ ЛЕКЦІЙ
з навчальної дисципліни
«ВСТУП ДО ПРОГРАМУВАННЯ .NET та JAVA»
для студентів ОС «бакалавр» спеціальностей
121 Інженерія програмного забезпечення
125 Кібербезпека**

Луцьк – 2023

Схемоконспект лекцій з навчальної дисципліни «Вступ до програмування .NET та JAVA» для студентів ОС «бакалавр» спеціальностей 121 Інженерія програмного забезпечення та 125 Кібербезпека / уклад. Є.О. Башков, А.О. Нікітенко. – Луцьк: ДонНТУ, 2023. – 517 с.

Навчальна дисципліна "Вступ до програмування .NET та JAVA" є невід'ємною складовою сучасної підготовки студентів спеціальностей 121 «Інженерія програмного забезпечення» та 125 «Кібербезпека». У контексті стрімкого розвитку сфери ІТ, де висока конкуренція й постійна необхідність вдосконалення, знання основ програмування на платформах .NET та JAVA стає ключовим фактором успіху.

Сучасні технології базуються на програмних рішеннях, які використовують мови програмування .NET та JAVA. Знання цих мов необхідне для розробки різноманітних програмних продуктів, починаючи від веб-додатків і закінчуючи мобільними додатками. Ці навички дозволяють студентам не лише розуміти сучасні технології, а й активно брати участь у їх розвитку.

Вивчення "Вступу до програмування .NET та JAVA" допомагає студентам засвоїти основні принципи програмування, роботу з об'єктно-орієнтованим підходом, та оволодіти важливими інструментами розробки. Це стає важливою основою для подальшого вивчення більш складних аспектів програмування та роботи з різноманітними технологіями.

Укладачі: _____Є.О. Башков, професор, д. т. н., професор кафедри ПМІ

_____А.О. Нікітенко, асистент кафедри ПМІ

Рецензент: _____С.О. Ковальов, доцент, к. т. н., доцент кафедри електронної техніки

Відповідальний за випуск: Н.О. Маслова, доцент, к. т. н., зав. каф. ПМІ

Затверджено навчально-методичним відділом ДонНТУ, протокол №5 від 16.01.2024 р.

Розглянуто на засіданні кафедри прикладної математики та інформатики, протокол № 13 від 27.12.2023 р.

ЗМІСТ

ВСТУП.....	7
ТЕМА 1. ОГЛЯД ПЛАТФОРМ .NET ТА JAVA.....	8
1. Огляд платформи .NET.....	9
2. Що нового у .NET 7.....	12
3. Компоненти .NET 7.....	13
4. Особливості .NET 7.....	15
5. Огляд платформи Java.....	17
6. Особливості мови Java.....	18
7. Oracle JDK та OpenJDK.....	21
8. Типи платформ Java.....	22
ТЕМА 2. БАЗОВІ ЕЛЕМЕНТИ ТА КОНСТРУКЦІЇ ПРОГРАМУВАННЯ НА МОВІ JAVA.....	25
1. Типи даних.....	26
2. Введення з консолі.....	30
3. Арифметичні операції.....	32
4. Умовні вирази.....	33
5. Логічні операції.....	36
6. Перетворення базових типів.....	38
7. Явні та неявні перетворення.....	39
8. Умовні конструкції.....	45
9. Масиви.....	57
ТЕМА 3. МЕТОДИ У JAVA.....	66
1. Методи.....	67
2. Параметри методів.....	72
3. Оператор return. Результат методу.....	74
4. Перевантаження методів.....	78
5. Рекурсивні функції.....	80
ТЕМА 4. СТРУКТУРИ ДАНИХ. КОЛЕКЦІЇ В JAVA.....	85
1. Визначення структур даних.....	86

ЗМІСТ

2. Типи колекцій. Інтерфейс Collection.....	90
3. Клас ArrayList та інтерфейс List.....	98
4. Черги та клас ArrayDeque.....	101
5. Клас LinkedList.....	107
6. Інтерфейс Set і клас HashSet.....	109
7. Інтерфейси SortedSet та NavigableSet. Клас TreeSet.....	111
8. Інтерфейси Comparable та Comparator.....	116
9. Інтерфейси Map та HashMap.....	121
10. Ітератори.....	125
ТЕМА 5. ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ НА МОВІ JAVA.....	131
1. Парадигми програмування.....	132
2. Класи та об'єкти.....	134
3. Конструктори.....	137
4. Пакети.....	142
5. Модифікатори доступу. Інкапсуляція.....	146
6. Статичні члени та модифікатор static.....	150
7. Об'єкти як параметри методів.....	154
8. Внутрішні та вкладені класи.....	155
9. Успадкування.....	158
10. Ієрархія успадкування та перетворення типів.....	166
11. Клас Object і його методи.....	171
12. Поліморфізм.....	175
13. Додаткова інформація.....	178
ТЕМА 6. АБСТРАКТНІ КЛАСИ ТА ІНТЕРФЕЙСИ.....	182
1. Абстрактні класи.....	183
2. Інтерфейси.....	188
3. Інтерфейси в механізмі зворотного виклику.....	204
4. Перерахування enum.....	209
5. Узагальнення (Generics).....	214
6. Обмеження узагальнень.....	226

ЗМІСТ

ТЕМА 7. ОБРОБКА РЯДКІВ.....	234
1. Введення в рядки. Клас String.....	235
2. Основні операції з рядками.....	243
3. StringBuffer та StringBuilder.....	255
4. Регулярні вирази.....	264
ТЕМА 8. ОБРОБКА ВИКЛЮЧЕНЬ.....	277
1. Введення в обробку виключень.....	278
2. Оператор throws.....	285
3. Автоматичне управління ресурсами.....	287
4. Класи виключень.....	289
5. Як JVM обробляє виключення.....	292
6. Створення користувальницьких виключень.....	293
ТЕМА 9. КЛАСИ СТАНДАРТНОЇ БІБЛІОТЕКИ (ДОДАТКОВІ КЛАСИ).....	298
1. Стандартна бібліотека Java.....	299
2. Математичні обчислення і клас Math.....	301
3. Великі числа BigInteger та BigDecimal.....	304
4. Робота з датами.....	307
5. Класи обгортки.....	309
ТЕМА 10. РОБОТА З ФАЙЛОВОЮ СИСТЕМОЮ ТА ПОТОКАМИ ВВЕДЕННЯ- ВИВЕДЕННЯ. СЕРІАЛІЗАЦІЯ ОБ'ЄКТІВ.....	315
1. Ознайомлення з потоками введення-виведення.....	316
2. Читання та запис файлів.....	323
3. Читання та запис байтових масивів.....	328
4. Буферізовані потоки.....	332
5. Форматоване введення та виведення.....	334
6. Читання та запис примітивних типів даних.....	339
7. Читання та запис текстових файлів.....	342
8. Буферізація символічних потоків.....	346
9. Серіалізація.....	348
10. Клас File. Робота з файлами та каталогами.....	354

ЗМІСТ

11. Робота з ZIP-архівами.....	358
ТЕМА 11. ВВЕДЕННЯ В STREAM API ТА ЛЯМБДА-ВИРАЗИ.....	363
1. Лямбда-вирази.....	364
2. Вбудовані функціональні інтерфейси.....	379
3. Stream API.....	382
4. Тип Optional.....	406
5. Паралельні потоки.....	418
ТЕМА 12. БАГАТОПОТОКОВЕ ПРОГРАМУВАННЯ.....	424
1. Введення в багатопотокове програмування. Клас Thread.....	425
2. Створення, виконання, завершення та переривання потоків.....	429
3. Синхронізація потоків. Оператор synchronized.....	440
4. Взаємодія потоків. Методи wait та notify.....	445
5. Семафори.....	450
6. Обмін між потоками. Клас Exchanger.....	455
7. Клас Phaser.....	457
8. Блокування. ReentrantLock.....	461
9. Умови блокування.....	464
ТЕМА 13. РОБОТА З БД. ОЗНАЙОМЛЕННЯ З JDBC.....	470
1. Введення в бази даних.....	471
2. SQL.....	473
3. Java та бази даних. JDBC.....	474
4. Відмінність між Statement та PreparedStatement.....	490
ТЕМА 14. ОСНОВИ ТЕСТУВАННЯ В JAVA.....	495
1. Введення в тестування програмного забезпечення.....	496
2. Типи тестування програмного забезпечення.....	498
3. Рівні тестування програмного забезпечення.....	501
4. Модульне тестування. Фреймворк JUnit.....	502
5. Анотації в JUnit.....	504
6. Приклад роботи JUnit.....	508
7. Test-driven development.....	511

ВСТУП

Дисципліна «Вступ до програмування .NET та JAVA» вводить студентів у захоплюючий світ створення програм, де вони отримають можливість не лише освоїти мови програмування, але й розвинути аналітичне мислення та креативний підхід до вирішення завдань. Вивчення .NET та JAVA дозволяє студентам оволодіти сучасними технологіями, які використовуються у розробці веб-додатків та корпоративних систем. Дисципліна ставить перед студентами завдання освоїти не лише технічні аспекти програмування, а й вирішувати завдання, пов'язані з ефективністю та оптимізацією програмного коду.

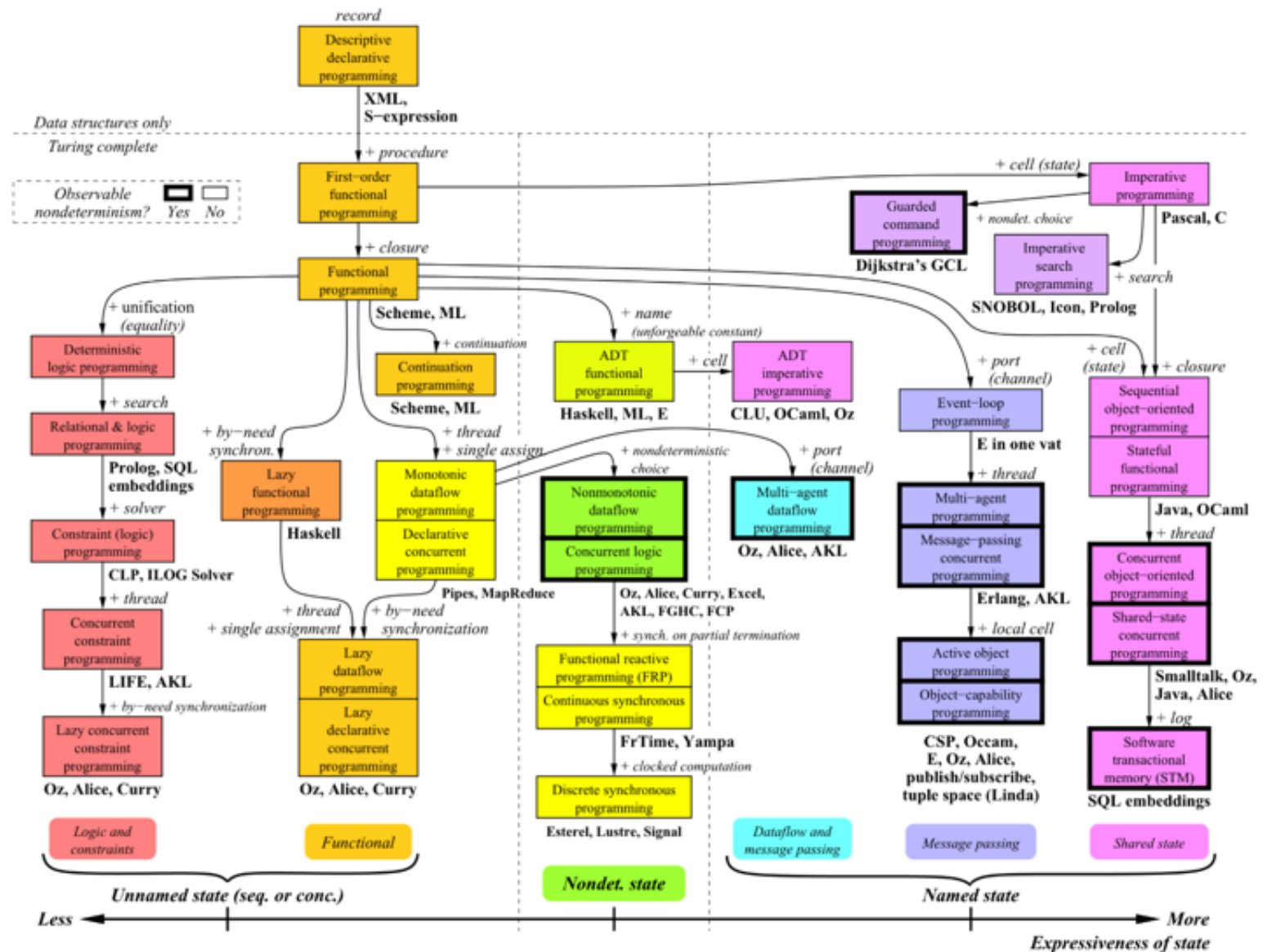
Знання та навички, отримані на цьому курсі, є основою для подальшого, більш ґрунтовного опанування спеціалізації Java розробника, а саме з технологіями баз даних (JDBC, JPA, Hibernate), створення клієнт-серверних додатків, додатків для Android, J2EE Web J2EE технології, а також зі сучасними фреймворками (Spring, Maven, JUnit).

.NET та JAVA #1

Огляд платформ .NET та JAVA

1. Огляд платформи .NET.
2. Що нового у .NET 7
3. Компоненти .NET 7
4. Особливості .NET 7
5. Огляд платформи Java
6. Особливості мови Java
7. Oracle JDK та OpenJDK
8. Типи платформ Java

Огляд платформи .NET



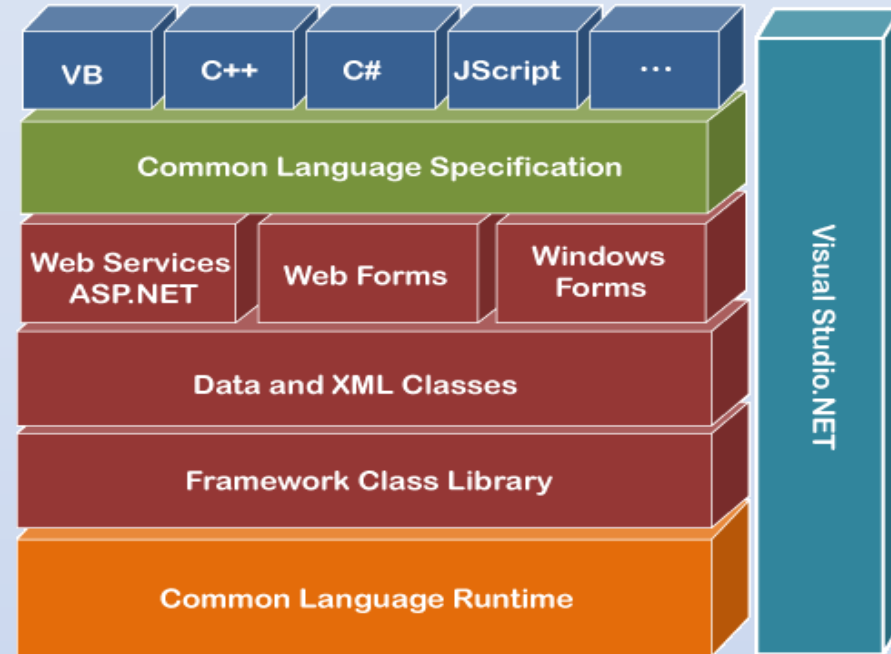
Огляд платформи .NET

.NET - це безкоштовна платформа для розробки програмного забезпечення з відкритим кодом, розроблена компанією Microsoft. Вона забезпечує основу для створення різноманітних додатків, включаючи десктопні, веб-, мобільні, ігрові та IoT-додатки. Платформа включає в себе набір бібліотек, середовище виконання та інструменти розробки, які дозволяють розробникам створювати додатки з використанням різних мов програмування, таких як **C#**, **F#** та Visual Basic.

.NET покликана спростити розробку програмного забезпечення, надаючи узгоджену модель програмування та загальний набір API, які можна використовувати на різних операційних системах та пристроях. Це дозволяє розробникам писати код один раз і розгортати його на різних платформах, включаючи Windows, macOS та Linux. Деякі з ключових компонентів платформи .NET включають **Common Language Runtime (CLR)**, бібліотеку класів **.NET Framework (FCL)** та інтегроване середовище розробки **Visual Studio (IDE)**.

Огляд платформи .NET

CLR відповідає за керування виконанням коду .NET та надання різних послуг, таких як керування пам'яттю, безпека типів та обробка винятків. **FCL** - це набір багаторазових класів і типів, які можна використовувати для створення різних додатків. А **Visual Studio IDE** - це надійне середовище розробки, яке надає різноманітні інструменти та функції, що допомагають розробникам писати, тестувати та налагоджувати **.NET** додатки.



Архітектура платформи .NET

Що нового у .NET 7

.NET 7 - це остання версія платформи .NET (на момент написання), випущена корпорацією Майкрософт у листопаді 2021 року. Вона містить кілька нових функцій та покращень у порівнянні з попередньою версією, **.NET 6**. Ось деякі з основних нововведень:

- 1) Підтримка Apple Silicon: **.NET 7** включає підтримку Apple Silicon, що дозволяє розробникам створювати та запускати додатки .NET на комп'ютерах Mac з процесорами Apple M1;
- 2) Спрощений спільний доступ до коду та пакетів: **.NET 7** представляє нову функцію під назвою Source Link, яка спрощує спільний доступ до коду та пакетів, дозволяючи розробникам переходити до вихідного коду пакета безпосередньо з IDE;
- 3) Покращена продуктивність: **.NET 7** включає в себе кілька покращень продуктивності, в тому числі скорочення часу запуску та зменшення використання пам'яті;
- 4) Покращена підтримка хмарної розробки: **.NET 7** включає в себе поліпшення для створення хмарних додатків, такі як поліпшена підтримка Kubernetes і Azure Functions;
- 5) Можливості мови C# 10: **.NET 7** підтримує нові можливості мови C# 10, такі як глобальне використання, файлові простори імен та інтерпольовані рядки як елементи формату.

Компоненти .NET 7

Нижче наведено список компонентів .NET:

- 1) Фреймворки та бібліотеки додатків;
- 2) Стандарт .NET;
- 3) Компоненти виконання;
- 4) Мовні компілятори;
- 5) Мови - C#, F# та Visual Basic;
- 6) Інструменти - **Visual Studio, Visual Studio for Mac, Visual Studio Code** та інтерфейс командного рядка (CLI) - інструменти, що використовуються для створення, тестування та розгортання .NET додатків.

Компоненти .NET 7

.NET можна використовувати для розробки наступних типів додатків:

- Десктопні додатки;
- Веб-додатки;
- Blazor;
- Мобільні додатки;
- Хмарні додатки;
- Ігри;
- Інтернет речей.

Особливості .NET 7

Кілька ключових особливостей .NET роблять її популярною та потужною платформою для розробки програмного забезпечення. Ось деякі з них:

- Крос-платформна розробка;
- Мовна сумісність;
- Common Language Runtime (CLR);
- Бібліотека базових класів (BCL);
- Компіляція Just-In-Time (JIT);
- Збір сміття;
- Інтегроване середовище розробки (IDE);
- .NET та DevOps.

Особливості .NET 7

Нативною (природною, природною) для платформи **.NET** є спеціально розроблена в 1998-2001 рр. фірмою Microsoft об'єктно-орієнтована мова C# (C Sharp). Сьогодні мова програмування C# є однією з найпотужніших, найдинамічніших і найзатребуваніших мов в ІТ-індустрії. Зараз вона використовується для написання додатків від невеликих настільних програм до великих веб-порталів і веб-сервісів, які обслуговують мільйони користувачів щодня.

C# є об'єктно-орієнтованою і в цьому відношенні багато чому навчилася від **Java** та **C++**. Наприклад, C# підтримує поліморфізм, успадкування, перевантаження операторів та статичну типізацію. Її об'єктно-орієнтований підхід дозволяє вирішувати завдання побудови великих, але гнучких, масштабованих і розширюваних додатків. Крім того, C# продовжує активно розвиватися і з кожним новим релізом додається все більше цікавих можливостей.

Огляд платформи JAVA

Сьогодні **Java** є однією з найпоширеніших мов програмування. Перша версія мови з'явилася в 1996 році в компанії **Sun Microsystems**, а пізніше була придбана компанією **Oracle**. Java була розроблена як універсальна мова програмування, яку можна було б використовувати для вирішення різноманітних завдань. Зараз Java пройшла довгий шлях розвитку, і було випущено різні версії. Поточна версія - **Java 20**, випущена в **березні 2023 року**. Java перетворилася з просто універсальної мови на цілу платформу та екосистему, яка об'єднує різні технології, що використовуються для різних завдань, від побудови десктопних додатків до створення масштабних веб-порталів та сервісів. Крім того, Java активно використовується для створення програмного забезпечення для найрізноманітніших пристроїв, включаючи звичайні комп'ютери, планшети, смартфони і мобільні телефони, і навіть побутову електроніку. Досить згадати популярність мобільної операційної системи **Android**, більшість додатків якої написані на Java.



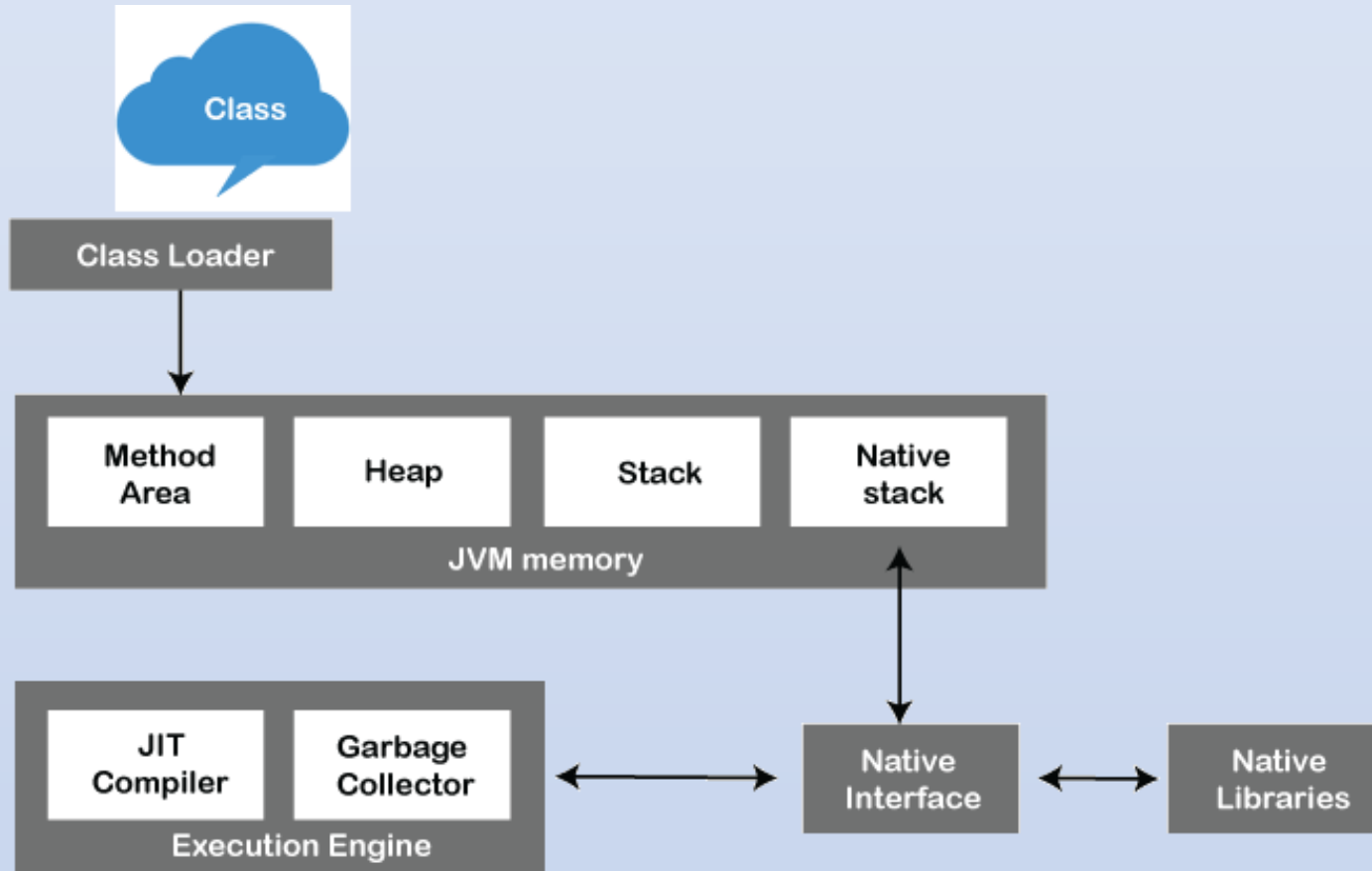
Особливості мови JAVA

Головна особливість мови Java полягає в тому, що код спочатку транслюється в спеціальний незалежний від платформи **байт-код**. Цей **байт-код** потім виконується **JVM (віртуальною машиною Java)**. У цьому відношенні Java відрізняється від стандартних інтерпретованих мов, таких як PHP і Perl. Java також не є повністю скомпільованою мовою, як C або C++.

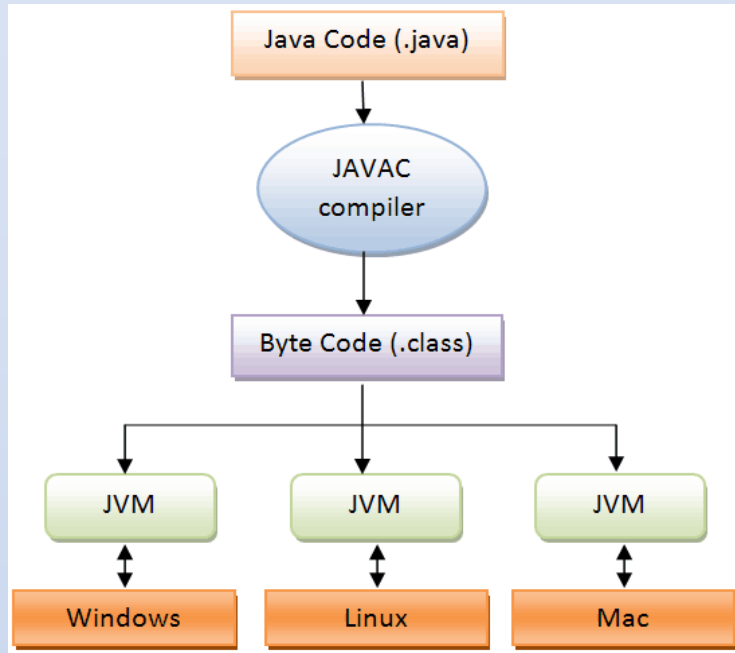
Така архітектура забезпечує крос-платформну сумісність і апаратну переносимість програм на Java, дозволяючи запускати схожі програми на різних платформах, таких як Windows, Linux і Mac OS без перекомпіляції. Кожна платформа може мати власну реалізацію JVM, але кожна може виконувати той самий код.

Віртуальна машина Java (JVM) - це рушій, який забезпечує середовище виконання для запуску коду Java або додатків. Він перетворює байт-код Java у машинну мову. JVM є частиною середовища виконання Java (JRE). В інших мовах програмування компілятор створює машинний код для конкретної системи. Однак компілятор Java створює код для віртуальної машини, відомої як Java Virtual Machine.

Особливості мови JAVA



Особливості мови JAVA



Робота Java Virtual Machine (**JVM**)

Ще однією важливою особливістю Java є підтримка автоматичного збору сміття. Це означає, що немає необхідності вручну звільняти пам'ять від раніше використаних об'єктів, як у C++.

Java - це об'єктно-орієнтована мова. Вона підтримує поліморфізм, успадкування та статичну типізацію. Об'єктно-орієнтований підхід вирішує проблему побудови великих, але в той же час гнучких, масштабованих і розширюваних додатків.

Oracle JDK та OpenJDK

Розробка на мові програмування Java вимагає спеціального набору інструментів, який називається **JDK** або **Java Development Kit**. Однак варто зазначити, що існують різні реалізації **JDK**, хоча вони використовують одну і ту ж мову Java. Дві найпопулярніші реалізації - Oracle JDK та OpenJDK. Які відмінності між ними?

Oracle JDK був повністю розроблений корпорацією Oracle, в той час як **OpenJDK** був розроблений спільно Oracle та кількома іншими компаніями.

Найбільша різниця між ними полягає в **ліцензуванні**. Згідно з ліцензією Oracle JDK, він є безкоштовним для особистого використання, а також для розробки, тестування та демонстрації додатків. В інших випадках (наприклад, для підтримки) потрібна комерційна ліцензія на основі підписки. OpenJDK, однак, є повністю безкоштовним.

З точки зору можливостей та наборів функцій, Oracle JDK та OpenJDK майже ідентичні. Однак відзначається, що Oracle JDK трохи швидший за OpenJDK, коли мова йде про продуктивність. Також деякі розробники стверджують, що OpenJDK трохи більш багатий на помилки, а Oracle JDK більш стабільний.

Типи платформ JAVA

Існує чотири різних типи платформ мови програмування Java:

1. **Java SE (Standard Edition)** – пропонує основну функціональність мови програмування Java. Він визначає всі основи типів та об'єктів до класів високого рівня. Він використовується для роботи в мережі, безпеки, доступу до баз даних, розробки графічного інтерфейсу користувача (GUI) та розбору XML. Саме з нього починається вивчення Java;
2. **Java EE (Enterprise Edition)** - набір технологій і специфікацій для створення програмного забезпечення рівня підприємства. Це видання призначене для створення серверних і розподілених компонентів.
3. **Java ME (Micro Edition)** - створена для використання в пристроях, обмежених по обчислювальній потужності, наприклад в мобільних телефонах, КПК, вбудованих системах;
4. **JavaFX** – технологія створення додатків з розвиненим графічним інтерфейсом користувача, в тому числі RIA (Rich Internet Applications) – корпоративних і бізнес-web-додатків;
5. **Java Card** – технологія розробки додатків, що працюють на смарт-картах та інших пристроях з дуже обмеженим обсягом пам'яті і слабкими можливостями обробки.

Контрольні питання

1. Надайте визначення платформі .NET;
2. Опишіть компоненти .NET;
3. Опишіть особливості .NET;
4. Надайте визначення платформі Java;
5. Проаналізуйте основні властивості мови програмування Java;
6. Яка відмінність між Oracle JDK та OpenJDK;
7. Поясніть, що означає термін "JVM" (Java Virtual Machine);
8. Визначте різновиди платформ в середовищі Java.

The END
Lec 1.

.NET та JAVA #2

Базові елементи та конструкції програмування на мові Java

1. Типи даних
2. Введення з консолі
3. Арифметичні операції
4. Умовні вирази
5. Логічні операції
6. Перетворення базових типів
7. Явні та неявні перетворення
8. Умовні конструкції
9. Масиви

Типи даних

Однією з основних особливостей **Java** є те, що ця мова є строго типізованою. А це означає, що кожна змінна і константа представляє певний тип і цей тип суворо визначений. Тип даних визначає діапазон значень, які може зберігати змінна або константа. Які є типи даних:

- **boolean**: зберігає значення true або false;
- **byte**: зберігає ціле число від -128 до 127 і займає 1 байт;
- **short**: зберігає ціле число від -32768 до 32767 і займає 2 байти;
- **int**: зберігає ціле число від -2147483648 до 2147483647 і займає 4 байти;
- **long**: зберігає ціле число від -9 223 372 036 854 775 808 до 9 223 372 036 854 775 807 і займає 8 байт;
- **double**: зберігає число з плаваючою крапкою від $\pm 4.9 \cdot 10^{-324}$ до $\pm 1.7976931348623157 \cdot 10^{308}$ і займає 8 байт. Як роздільник цілої та дробової частини в дробових літералах використовується крапка;
- **float**: зберігає число з плаваючою крапкою від $-3.4 \cdot 10^{38}$ до $3.4 \cdot 10^{38}$ і займає 4 байти;
- **char**: зберігає одиночний символ у кодуванні UTF-16 і займає 2 байти, тому діапазон збережених значень від 0 до 65535.

Типи даних

Цілі числа

Усі цілочисельні літерали, наприклад, числа 10, 4, -5, сприймаються як значення типу `int`, однак ми можемо присвоювати цілочисельні літерали іншим цілочисельним типам: `byte`, `long`, `short`. У цьому випадку Java автоматично здійснює відповідні перетворення:

```
byte a = 1;
```

```
short b = 2;
```

```
long c = 2121;
```

Однак якщо ми захочемо присвоїти змінній типу `long` дуже велике число, яке виходить за межі допустимих значень для типу `int`, то ми зіткнемося з помилкою під час компіляції:

```
long num = 2147483649;
```

Тут число 2147483649 є допустимим для типу `long`, але виходить за граничні значення для типу `int`. І оскільки всі цілочисельні значення за замовчуванням розцінюються як значення типу `int`, то компілятор вкаже нам на помилку. Щоб розв'язати проблему, треба додати до числа суфікс `l` або `L`, який вказує, що число представляє тип `long`.

Типи даних

Числа з плаваючою крапкою

Під час присвоєння змінній типу `float` дробового літерала з плаваючою крапкою, наприклад, `3.1`, `4.5` тощо, Java автоматично розглядає цей літерал як значення типу `double`. І щоб вказати, що дане значення має розглядатися як `float`, нам треба використовувати суфікс `f`:

```
float f1 = 30.6f;
```

```
double db = 30.6;
```

І хоча в цьому випадку обидві змінні мають практично одне значення, але ці значення будуть по-різному розглядатися і займатимуть різне місце в пам'яті.

Типи даних

Символи та рядки

Як значення змінна символьного типу отримує одиночний символ, укладений в одинарні лапки: `char ch='e';`. Крім того, змінній символьного типу також можна присвоїти цілочисельне значення від 0 до 65535. У цьому випадку змінна знову ж таки зберігатиме символ, а цілочисельне значення вказуватиме на номер символу в таблиці символів Unicode (UTF-16). Наприклад:

```
char ch=102; // символ 'f'  
System.out.println(ch);
```

Ще однією формою задавання символьних змінних є шістнадцяткова форма: змінна отримує значення в шістнадцятковій формі, яке слідує після символів `"\u"`. Наприклад, `char ch='\u0066';` знову ж таки зберігатиме символ 'f'.

Символьні змінні не варто плутати з рядковими, 'a' не ідентично "a". Рядкові змінні представляють об'єкт `String`, який на відміну від `char` або `int` не є примітивним типом у Java:

```
String hello = "Hello...";  
System.out.println(hello);
```

Введення з консолі

Для отримання введення з консолі в класі **System** визначено об'єкт **in**. Однак безпосередньо через об'єкт **System.in** не дуже зручно працювати, тому, як правило, використовують клас **Scanner**, який, у свою чергу, використовує **System.in**. Наприклад, напишемо маленьку програму, яка здійснює введення чисел:

```
import java.util.Scanner;

public class Program {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.print("Input a number: ");
        int num = in.nextInt();
        System.out.printf("Your number: %d \n", num);
        in.close();
    }
}
```

Для створення самого об'єкта **Scanner** у його конструктор передається об'єкт **System.in**. Після цього ми можемо отримувати значення, що вводяться. Наприклад, у цьому випадку спочатку виводимо запрошення до введення і потім отримуємо число, що вводиться, у змінну **num**.

Введення з консолі

Клас **Scanner** має низку методів, які дають змогу отримати введені користувачем значення:

- **next()**: зчитує введений рядок до першого пробілу;
- **nextLine()**: зчитує весь введений рядок;
- **nextInt()**: зчитує введене число int;
- **nextDouble()**: зчитує введене число double;
- **nextBoolean()**: зчитує значення boolean;
- **nextByte()**: зчитує введене число byte;
- **nextFloat()**: зчитує введене число float;
- **nextShort()**: зчитує введене число short;

Тобто для введення значень кожного примітивного типу в класі **Scanner** визначено свій метод.

Арифметичні операції

В **арифметичних операціях** беруть участь числа. У Java є бінарні арифметичні операції (проводяться над двома операндами) і унарні (виконуються над одним операндом). До бінарних операцій відносять такі: +операція додавання двох чисел; -операція віднімання двох чисел; *операція множення двох чисел; /операція ділення двох чисел.

Під час ділення варто враховувати, що якщо в операції беруть участь два цілих числа, то результат ділення буде округлятися до цілого числа, навіть якщо результат присвоюється змінній float або double:

Також є дві унарні арифметичні операції, які виконуються над одним числом: ++ (інкремент) і -- (декремент). Кожна з операцій має два різновиди: **префіксний і постфіксний**;

++ (префіксний інкремент) - передбачає збільшення змінної на одиницю, наприклад, `z=++y` (спочатку значення змінної `y` збільшується на 1, а потім її значення присвоюється змінній `z`);

-- (префіксний декремент) - зменшення змінної на одиницю, наприклад, `z=--y` (спочатку значення змінної `y` зменшується на 1, а потім її значення присвоюється змінній `z`);

-- (постфіксний декремент) - `z=y--` (спочатку значення змінної `y` присвоюється змінній `z`, а потім значення змінної `y` зменшується на 1).

Умовні вирази

Умовні вирази являють собою деяку умову і повертають значення типу `boolean`, тобто значення `true` (якщо умова істинна), або значення `false` (якщо умова хибна). До них відносять:

- 1) Операції порівняння
- 2) Логічні операції

В операціях **порівняння** порівнюються два операнди, і повертається значення типу `boolean` - `true`, якщо вираз є правильним, і `false`, якщо вираз є неправильним:

1) `==` порівнює два операнди на рівність і повертає `true` (якщо операнди рівні) і `false` (якщо операнди не рівні):

```
int a = 10;  
int b = 4;  
boolean c = a == b;           // false  
boolean d = a == 10;          // true
```

Умовні вирази

2) `!=` порівнює два операнди і повертає `true`, якщо операнди НЕ рівні, і `false`, якщо операнди рівні:

```
int a = 10;  
int b = 4;  
boolean c = a != b;           // true  
boolean d = a != 10;          // false
```

3) `<` (менше ніж) - повертає `true`, якщо перший операнд менший за другий, інакше повертає `false`:

```
int a = 10;  
int b = 4;  
boolean c = a < b;           // false
```

Умовні вирази

4) **>** (більше ніж) - повертає true, якщо перший операнд більший за другий, інакше повертає false:

```
int a = 10;
```

```
int b = 4;
```

```
boolean c = a > b;    // true
```

5) **>=** (більше або дорівнює) – повертає true, якщо перший операнд більший за другий або дорівнює другому, інакше повертає false:

```
boolean c = 10 >= 10;    // true
```

6) **<=** (менше або дорівнює) - повертає true, якщо перший операнд менший за другий або дорівнює другому, інакше повертає false

```
boolean c = 10 <= 10;    // true
```

Логічні операції

Також у Java є **логічні операції**, які також представляють умову та повертають true або false і зазвичай об'єднують кілька операцій порівняння. До логічних операцій відносять такі:

- 1) **|** - $c = a | b$; (с дорівнює true, якщо або a, або b (або і a, і b) дорівнюють true, інакше с дорівнюватиме false)
- 2) **&** - $c = a \& b$; (с дорівнює true, якщо і a, і b дорівнюють true, інакше с дорівнюватиме false)

Логічні операції

Тут у нас дві пари операцій `|` і `||` (а також `&` і `&&`) виконують схожі дії, проте ж вони не рівнозначні.

Вираз `c=a|b`; обчислюватиме спочатку обидва значення - `a` та `b` і на їхній основі виводитиме результат.

У виразі ж `c=a||b`; спочатку обчислюватиметься значення `a`, і якщо воно дорівнюватиме `true`, то обчислення значення `b` вже сенсу не має, оскільки в нас у будь-якому разі вже `c` дорівнюватиме `true`. Значення `b` буде обчислюватися тільки в тому випадку, якщо `a` дорівнює `false`

Те ж саме стосується пари операцій `&` і `&&`. У виразі `c=a&b`; обчислюватимуться обидва значення - `a` і `b`.

У виразі ж `c=a&&b`; спочатку обчислюватиметься значення `a`, і якщо воно дорівнюватиме `false`, то обчислення значення `b` уже не має сенсу, оскільки значення `c` у будь-якому разі дорівнюватиме `false`. Значення `b` буде обчислюватися тільки в тому випадку, якщо `a` дорівнює `true`

Таким чином, операції `||` і `&&` зручніші в обчисленнях, даючи змогу скоротити час на обчислення значення виразу і тим самим підвищуючи продуктивність. А операції `|` і `&` більше підходять для виконання порозрядних операцій над числами.

Перетворення базових типів

Кожен базовий тип даних займає певну кількість байт пам'яті. Це накладає обмеження на операції, до яких залучені різні типи даних. Розглянемо такий приклад:

```
int a = 4;  
byte b = a; // ! Помилка
```

Хоча і тип `byte`, і тип `int` представляють цілі числа. Ба більше, значення змінної `a`, яке присвоюється змінній типу `byte`, цілком укладається в діапазон значень для типу `byte` (від -128 до 127). Проте ми стикаємося з помилкою на етапі компіляції. Оскільки в даному випадку ми намагаємося присвоїти деякі дані, які займають 4 байти, змінній, яка займає всього один байт.

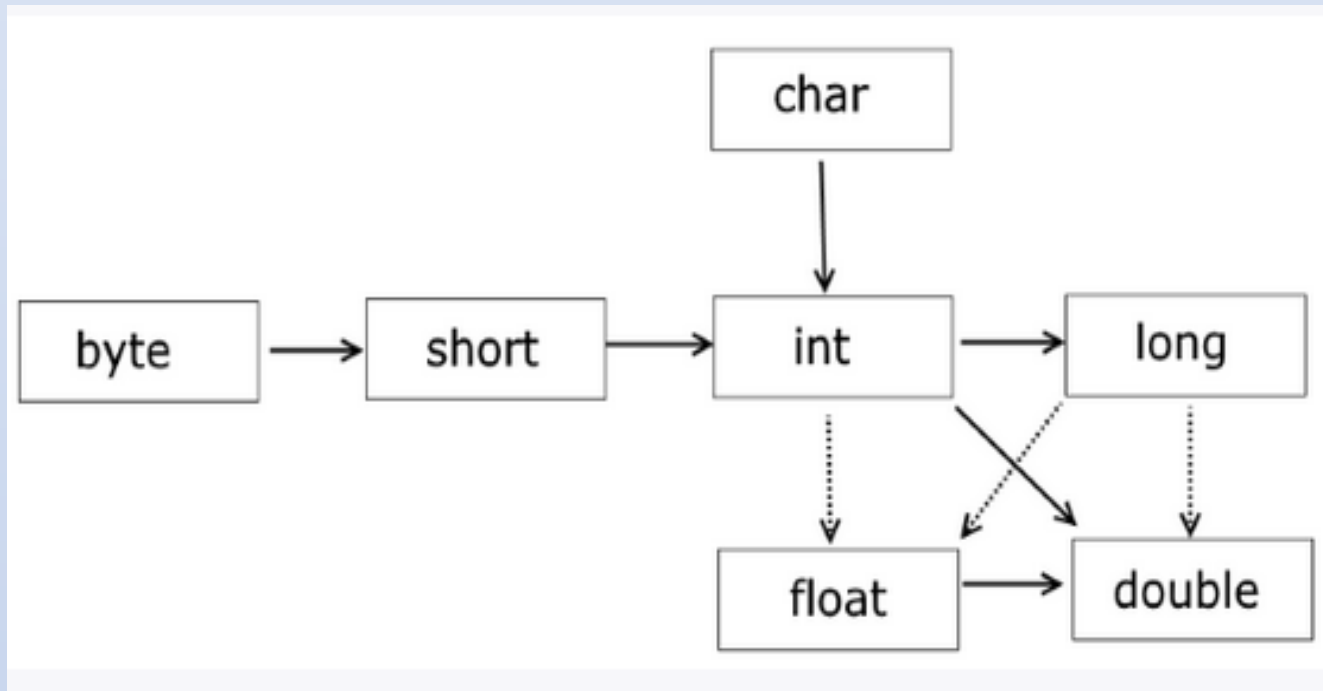
Проте в програмі може знадобитися, щоб подібне перетворення було виконано. У цьому випадку необхідно використовувати операцію перетворення типів (операція `()`):

```
int a = 4;  
byte b = (byte)a; // перетворення типів: від типу int до типу byte  
System.out.println(b); // 4
```

Операція перетворення типів передбачає зазначення в дужках того типу, до якого треба перетворити значення. Наприклад, у випадку операції `(byte)a`, відбувається перетворення даних типу `int` у тип `byte`. У підсумку ми отримаємо значення типу `byte`.

Явні та неявні перетворення

Коли в одній операції залучені дані різних типів, не завжди необхідно використовувати операцію **перетворення типів**. Деякі види перетворень виконуються неявно, автоматично.



Автоматичне перетворення

Явні та неявні перетворення

Стрілками на рисунку показано, які перетворення типів можуть виконуватися автоматично. Пунктирними стрілками показано автоматичні перетворення з втратою точності.

Автоматично без будь-яких проблем виконуються розширювальні перетворення (**widening**) - вони розширюють представлення об'єкта в пам'яті. Наприклад:

```
byte b = 7;
```

```
int d = b; // перетворення від byte до int
```

Автоматичні перетворення, що розширюють, представлені такими ланцюжками:

```
byte -> short -> int -> long
```

```
int -> double
```

```
short -> float -> double
```

```
char -> int
```

Явні та неявні перетворення

Деякі перетворення можуть здійснюватися автоматично між типами даних однакової розрядності або навіть від типу даних з більшою розрядністю до типу з меншою розрядністю. Це такі ланцюжки перетворень:

```
int -> float, long -> float i long -> double
```

Вони виконуються без помилок, але при перетворенні ми можемо зіткнутися з втратою інформації.

Наприклад:

```
int a = 2147483647;  
float b = a; // від типу int до типу float  
System.out.println(b); // 2.14748365E9
```

Явні та неявні перетворення

У всіх інших перетвореннях примітивних типів явним чином застосовується операція перетворення типів. Зазвичай це звужувальні перетворення (narrowing) від типу з більшою розрядністю до типу з меншою розрядністю:

```
long a = 4;
```

```
int b = (int) a;
```

При застосуванні явних перетворень ми можемо зіткнутися з втратою даних. Наприклад, у наступному коді у нас не виникне жодних проблем:

```
int a = 5;
```

```
byte b = (byte) a;
```

```
System.out.println(b);      // 5
```

Число 5 цілком укладається в діапазон значень типу byte, тому після перетворення змінна b дорівнюватиме 5. Але що буде в наступному випадку:

```
int a = 258;
```

```
byte b = (byte) a;
```

```
System.out.println(b);      // 2
```

Явні та неявні перетворення

Результатом буде число 2. У цьому випадку число 258 поза діапазоном для типу byte (від -128 до 127), тому відбудеться усічення значення. Число a, яке дорівнює 258, у двійковій системі дорівнюватиме 00000000 00000000 00000001 00000010. Значення типу byte займають у пам'яті тільки 8 біт. Тому двійкове подання числа int усікається до 8 правих розрядів, тобто 00000010, що в десятковій системі дає 2.

При перетворенні значень з плаваючою крапкою до цілочисельних значень, відбувається усічення дробової частини:

```
double a = 56.9898;
```

```
int b = (int)a;
```

Тут значення числа b дорівнюватиме 56, незважаючи на те, що число 57 було б ближчим до 56.9898. Щоб уникнути подібних казусів, треба застосовувати функцію округлення, яка є в математичній бібліотеці Java:

```
double a = 56.9898;
```

```
int b = (int)Math.round(a);
```

Явні та неявні перетворення

Непоодинокі ситуації, коли доводиться застосовувати різні операції, наприклад, додавання і добуток, над значеннями різних типів. Тут також діють деякі правила:

- якщо один з операндів операції належить до типу `double`, то і другий операнд перетворюється до типу `double`;
- якщо попередня умова не дотримана, а один з операндів операції належить до типу `float`, то і другий операнд перетворюється до типу `float`;
- якщо попередні умови не дотримано, один з операндів операції відноситься до типу `long`, то і другий операнд перетворюється до типу `long`;
- інакше всі операнди операції перетворюються до типу `int`.

Умовні конструкції

Одним із фундаментальних елементів багатьох мов програмування є умовні конструкції. Ці конструкції дають змогу спрямувати роботу програми одним зі шляхів залежно від певних умов.

У мові Java використовуються такі умовні конструкції: **if..else** і **switch..case**

Конструкція if/else

Вираз if/else перевіряє істинність деякої умови і залежно від результатів перевірки виконує певний код:

```
int num1 = 6;  
int num2 = 4;  
if(num1>num2){  
    System.out.println("Перше число більше другого");  
}
```

Після ключового слова if ставиться умова. І якщо ця умова виконується, то спрацьовує код, який поміщено далі в блоці if після фігурних дужок. В якості умов виступає операція порівняння двох чисел.

Умовні конструкції

Але що, якщо ми захочемо, щоб у разі недотримання умови також виконувалися якісь дії? У цьому випадку ми можемо додати блок `else`:

```
int num1 = 6;  
int num2 = 4;  
if(num1>num2) System.out.println("Перше число більше другого");  
else System.out.println("Перше число менше другого");
```

Але під час порівняння чисел ми можемо нарахувати три стани: перше число більше за друге, перше число менше за друге і числа рівні. За допомогою виразу `else if`, ми можемо обробляти додаткові умови:

```
int num1 = 6;  
int num2 = 8;  
if(num1>num2)System.out.println("Первое число больше второго");  
else if(num1<num2) System.out.println("Перше число більше другого");  
else System.out.println("Перше число менше другого");
```

Умовні конструкції

Також ми можемо з'єднати відразу кілька умов, використовуючи логічні оператори:

```
int num1 = 8;
```

```
int num2 = 6;
```

```
if(num1 > num2 && num1>7) System.out.println("Перше число більше другого та  
більше 7");
```

Тут блок if виконуватиметься, якщо `num1 > num2` дорівнює `true` і одночасно `num1>7` дорівнює `true`.

Умовні конструкції

Конструкція switch

Конструкція switch/case аналогічна конструкції if/else, оскільки дає змогу обробити одразу кілька умов:

```
int num = 8;
switch(num){
    case 1:
        System.out.println("число дорівнює 1");
        break;
    case 8:
        System.out.println("число дорівнює 8");
        num++;
        break;
    case 9:
        System.out.println("число дорівнює 9");
        break;
    default: System.out.println("число не дорівнює 1, 8, 9");}
```

Після ключового слова switch у дужках йде порівнюваний вираз. Значення цього виразу послідовно порівнюється зі значеннями, поміщеними після операторів case. І якщо збіг знайдено, то буде виконує відповідний блок case.

Умовні конструкції

Наприкінці блоку case ставиться оператор break, щоб уникнути виконання інших блоків. Наприклад, якби прибрали оператор break у такому випадку:

case 8:

```
System.out.println("число дорівнює 8");
```

```
num++;
```

case 9:

```
System.out.println("число дорівнює 9");
```

```
break;
```

то виконався б блок case 8, (оскільки змінна num дорівнює 8). Але оскільки в цьому блоці оператор break відсутній, то почав би виконуватися блок case 9.

Якщо ми хочемо також обробити ситуацію, коли збігу не буде знайдено, то можна додати блок default, як у прикладі вище. Хоча блок default необов'язковий.

Умовні конструкції

Також ми можемо визначити одну дію відразу для кількох блоків case поспіль:

```
int num = 3;
int output = 0;
switch(num){
    case 1:
        output = 3;
        break;
    case 2:
    case 3:
    case 4:
        output = 6;
        break;
    case 5:
        output = 12;
        break;
    default:
        output = 24;}
System.out.println(output);
```

Умовні конструкції

Тернарна операція

Тернарна операція має такий синтаксис: [перший операнд - умова] ? [другий операнд] : [третій операнд]. Таким чином, у цій операції беруть участь одразу три операнди. Залежно від умови тернарна операція повертає другий або третій операнд: якщо умова дорівнює true, то повертається другий операнд; якщо умова дорівнює false, то третій. Наприклад:

```
int x=3;
```

```
int y=2;
```

```
int z = x<y? (x+y) : (x-y);
```

```
System.out.println(z);
```

Тут результатом тернарної операції є змінна z. Спочатку перевіряється умова $x < y$. І якщо вона виконується, то z дорівнюватиме другому операнду - $(x+y)$, інакше z дорівнюватиме третьому операнду.

Умовні конструкції

Цикли

Ще одним видом керуючих конструкцій є цикли. Цикли дозволяють залежно від певних умов виконувати певну дію безліч разів. У мові Java є такі види циклів:

- for
- while
- do...while

Цикл for

Цикл for має таке формальне визначення:

```
for ([ініціалізація лічильника]; [умова]; [зміна лічильника])  
{  
    // дії  
}
```


Умовні конструкції

Розглянемо стандартний цикл for:

```
for (int i = 1; i < 9; i++){  
    System.out.printf("Квадрат числа %d дорівнює %d \n", i, i * i);  
}
```

Перша частина оголошення циклу - `int i = 1` створює та ініціалізує лічильник `i`. Лічильник необов'язково повинен представляти тип `int`. Це може бути і будь-який інший числовий тип, наприклад, `float`. Перед виконанням циклу значення лічильника дорівнюватиме 1. У цьому випадку це те ж саме, що й оголошення змінної.

Друга частина - умова, за якої буде виконуватися цикл. У цьому випадку цикл буде виконуватися, поки `i` не досягне 9.

І третя частина - збільшення лічильника на одиницю. Знову ж таки нам необов'язково збільшувати на одиницю. Можна зменшувати: `i--`.

У підсумку блок циклу спрацює 8 разів, поки значення `i` не стане рівним 9. І щоразу це значення буде збільшуватися на 1.

Умовні конструкції

Нам необов'язково вказувати всі умови під час оголошення циклу. Наприклад, ми можемо написати так:

```
int i = 1;
for (; ;){
    System.out.printf("Квадрат числа %d дорівнює %d \n", i, i * i);
}
```

Визначення циклу залишилося тим самим, тільки тепер блоки у визначенні в нас порожні: `for (; ;)`. Тепер немає ініціалізованої змінної-лічильника, немає умови, тому цикл працюватиме вічно - нескінченний цикл.

Або можна опустити низку блоків:

```
int i = 1;
for (; i<9;){
    System.out.printf("Квадрат числа %d дорівнює %d \n", i, i * i);
    i++;
}
```

Умовні конструкції

Цикл do

Цикл do спочатку виконує код циклу, а потім перевіряє умову в інструкції while. І поки ця умова істинна, цикл повторюється. Наприклад:

```
int j = 7;
do{
    System.out.println(j);
    j--;
}
while (j > 0);
```

У цьому випадку код циклу спрацює 7 разів, поки j не виявиться рівним нулю. Важливо зазначити, що цикл do гарантує хоча б одноразове виконання дій, навіть якщо умова в інструкції while не буде істинною.

```
int j = -1;
do{
    System.out.println(j);
    j--;
}
while (j > 0);
```

Хоча змінна j від початку менша за 0, цикл все одно один раз виконається.

Умовні конструкції

Цикл while

Цикл while відразу перевіряє істинність деякої умови, і якщо умова істинна, то код циклу виконується:

```
int j = 6;
while (j > 0){
    System.out.println(j);
    j--;
}
```

Оператори continue і break

Оператор break дозволяє вийти з циклу в будь-який його момент, навіть якщо цикл не закінчив свою роботу:

Наприклад:

```
for (int i = 0; i < 10; i++){
    if (i == 5)
        break;
    System.out.println(i);
}
```

Коли лічильник стане рівним 5, спрацює оператор break, і цикл завершиться.

Масиви

Масив являє собою набір однотипних значень. Оголошення масиву схоже на оголошення звичайної змінної, яка зберігає одиночне значення, причому є два способи оголошення масиву:

тип_даних назва_масиву[];

// або

тип_даних [] назва_масиву;

Наприклад, визначимо масив чисел:

```
int nums[];
```

```
int[] nums2;
```

Після оголошення масиву ми можемо ініціалізувати його:

Створення масиву здійснюється за допомогою такої конструкції: `new тип_даних[кількість_елементів]`, де `new` - ключове слово, що виділяє пам'ять для зазначеної в дужках кількості елементів. Наприклад, `nums = new int[4]`; - у цьому виразі створюється масив із чотирьох елементів `int`, і кожен елемент матиме значення за замовчуванням - число 0.

Масиви

Також можна відразу при оголошенні масиву ініціалізувати його:

```
int nums[] = new int[4];    // масив з 4 чисел
```

```
int[] nums2 = new int[5];   // масив з 5 чисел
```

При подібній ініціалізації всі елементи масиву мають значення за замовчуванням. Для числових типів (у тому числі для типу `char`) це число 0, для типу `boolean` це значення `false`, а для решти об'єктів це значення `null`. Наприклад, для типу `int` значенням за замовчуванням є число 0, тому вище визначений масив `nums` складатиметься з чотирьох нулів.

Однак також можна задати конкретні значення для елементів масиву при його створенні:

```
int[] nums = new int[] { 1, 2, 3, 5 };
```

```
int[] nums2 = { 1, 2, 3, 5 };
```

Варто зазначити, що в цьому випадку в квадратних дужках не вказується розмір масиву, оскільки він обчислюється за кількістю елементів у фігурних дужках.

Масиви

Після створення масиву ми можемо звернутися до будь-якого його елемента за індексом, який передається в квадратних дужках після назви змінної масиву:

```
int[] nums = new int[4];  
nums[0] = 1;  
nums[1] = 2;  
nums[2] = 4;  
nums[3] = 100;  
System.out.println(nums[2]);    // 4
```

Індексація елементів масиву починається з 0, тому в цьому випадку, щоб звернутися до четвертого елемента в масиві, нам треба використовувати вираз `nums[3]`.

І оскільки в нас масив визначений тільки для 4 елементів, то ми не можемо звернутися, наприклад, до шостого елемента: `nums[5] = 5;`. Якщо ми так спробуємо зробити, то ми отримаємо помилку.

Масиви

Довжина масиву

Найважливішою властивістю, яку мають масиви, є властивість `length`, що повертає довжину масиву, тобто кількість його елементів:

```
int[] nums = {1, 2, 3, 4, 5};  
int length = nums.length;    // 5
```

Нерідко буває невідомим останній індекс, і щоб отримати останній елемент масиву, ми можемо використовувати цю властивість:

```
int last = nums[nums.length-1];
```

Багатовимірні масиви

Раніше ми розглядали одновимірні масиви, які можна представити як ланцюжок або рядок однотипних значенбагатовимірними. Найвідоміший багатовимірний масив - таблиця, що представляє двовимірний ь. Але крім одновимірних масивів також бувають і масив:

```
int[] nums1 = new int[] { 0, 1, 2, 3, 4, 5 };  
int[][] nums2 = { { 0, 1, 2 }, { 3, 4, 5 } };
```


Масиви

Оскільки масив nums2 двовимірний, він являє собою просту таблицю. Його також можна було створити таким чином: `int[][] nums2 = new int[2][3];`. Кількість квадратних дужок вказує на розмірність масиву. А числа в дужках - на кількість рядків і стовпців. І також, використовуючи індекси, ми можемо використовувати елементи масиву в програмі:

```
// встановимо елемент першого стовпця другого рядка
```

```
nums2[1][0]=44;
```

```
System.out.println(nums2[1][0]);
```

Оголошення тривимірного масиву могло б виглядати так:

```
int[][][] nums3 = new int[2][3][4];
```

Зубчастий масив

Багатовимірні масиви можуть бути також представлені як "зубчасті масиви". У вищенаведеному прикладі двовимірний масив мав 2 рядки і три стовпці, тому в нас виходила рівна таблиця. Але ми можемо кожному елементу в двовимірному масиві присвоїти окремий масив із різною кількістю елементів:

```
int[][] nums = new int[3][];
```

```
nums[0] = new int[2];
```

```
nums[1] = new int[3];
```

```
nums[2] = new int[5];
```

Масиви

foreach

Спеціальна версія циклу for призначена для перебору елементів у наборах елементів, наприклад, у масивах і колекціях. Вона аналогічна дії циклу foreach , який є в інших мовах програмування. Формальне її оголошення:

```
for (тип_даних назва_змінної : контейнер) {  
    // дії  
}
```

Наприклад:

```
int[] array = new int[] { 1, 2, 3, 4, 5 };  
for (int i : array) {  
    System.out.println(i);  
}
```

Як контейнер у цьому випадку виступає масив даних типу int. Потім оголошується змінна з типом int.

Масиви

Водночас ця версія циклу for гнучкіша порівняно з for (int i : array). Зокрема, у цій версії ми можемо змінювати елементи:

```
int[] array = new int[] { 1, 2, 3, 4, 5 };  
for (int i=0; i<array.length;i++){  
    array[i] = array[i] * 2;  
    System.out.println(array[i]);  
}
```

Для перебору багатовимірної масиви в циклі спочатку створюється цикл для перебору за рядками, а потім усередині першого циклу створюється внутрішній цикл для перебору за стовпцями конкретного рядка. Подібним чином можна перебрати і тривимірні масиви та набори з великою кількістю розмірностей.

```
for (int i = 0; i < nums.length; i++){  
    for(int j=0; j < nums[i].length; j++){  
    }
```

Контрольні питання

1. Визначте поняття "тип даних" та перерахуйте типи даних, які існують у мові програмування Java;
2. Розкрийте поняття префіксного та постфіксного інкремента в мові програмування Java;
3. Розмежуйте префіксний та постфіксний декремент у контексті мови програмування Java;
4. Охарактеризуйте пріоритет арифметичних операцій у мові програмування Java;
5. Визначте операції порівняння, які використовуються в мові програмування Java;
6. Надайте визначення терміну "масив" у контексті програмування на Java;
7. Поясніть різницю між циклом `foreach` та циклом `for` в мові програмування Java;
8. Розкажіть про відмінності між циклом `do...while` та циклом `while` у мові програмування Java;
9. Розгляньте складну логіку управління циклом, включаючи використання операторів `continue` та `break` в мові програмування Java.

The END
Lec 2.

.NET та JAVA #3

Методи у Java

1. Методи
2. Параметри методів
3. Оператор return. Результат методу
4. Перевантаження методів
5. Рекурсивні функції

Методи

Метод в Java або Методи Java - це набір операторів, які виконують певну задачу і повертають результат тому, хто їх викликає. Метод Java може виконувати певну задачу, не повертаючи нічого. Методи Java дозволяють нам повторно використовувати код без повторного набору коду. У Java кожен метод повинен бути частиною деякого класу, що відрізняється від таких мов, як **C**, **C++** та **Python**.

1. Метод подібний до функції, тобто використовується для опису поведінки об'єкта;
2. Це набір кодів, які виконують певну задачу.

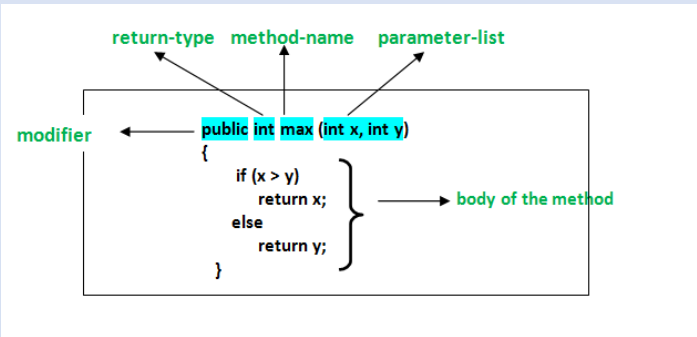
Синтаксис методу:

```
[модифікатори] тип_поверненого_значення назва_методу([параметри]){  
    // тіло методу  
}
```

Перевага методів:

- 1) Повторне використання коду;
- 2) Оптимізація коду.

Методи



Загалом, оголошення методів складаються з 6 компонентів:

1. Модифікатор: Визначає тип доступу до методу, тобто звідки до нього можна отримати доступ у вашій програмі. Всього є 4 модифікатора доступу;
2. Тип значення, що повертається: Тип даних значення, що повертається методом, або void, якщо метод не повертає значення. У синтаксисі є обов'язковим;
3. Ім'я методу: правила для імен полів застосовуються і до імен методів, але угода дещо відрізняється. Це поле є обов'язковим у синтаксисі;
4. Список параметрів: Список вхідних параметрів, розділених комами, з типом даних, що передують їм, у круглих дужках. Якщо параметрів немає, ви повинні використовувати порожні дужки (). У синтаксисі це необов'язково;

Методи

5. Список винятків: Винятки, які ви очікуєте, що метод може згенерувати, ви можете вказати ці винятки. Не є обов'язковим у синтаксисі;

6. Тіло методу: воно укладається між фігурними дужками. Код, який потрібно виконати для виконання запланованих операцій. Не є обов'язковим у синтаксисі.

Типи методів в Java

У мові Java існує два типи методів:

- 1. Визначений метод (Predefined Method).** В Java наперед визначені методи - це методи, які вже визначені в бібліотеках класів Java, відомі як наперед визначені методи. Він також відомий як стандартний метод бібліотеки або вбудований метод. Ми можемо безпосередньо використовувати ці методи, просто викликаючи їх у програмі в будь-який момент;
- 2. Метод, визначений користувачем.** Метод, написаний користувачем або програмістом, відомий як метод, визначений користувачем. Ці методи модифікуються відповідно до вимог.

Методи

За замовчуванням головний клас будь-якої програми на Java містить метод `main`, який слугує точкою входу в програму:

```
public static void main(String[] args) {  
    System.out.println("Hello, World!");  
}
```

Ключові слова `public` і `static` є модифікаторами. Далі йде тип значення, що повертається. Ключове слово `void` вказує на те, що метод нічого не повертає. Потім ідуть назва методу - `main` і в дужках параметри методу - `String[] args`. І у фігурні дужки укладено тіло методу - усі дії, які він виконує.

Створимо ще кілька методів:

```
public class Program{  
    public static void main (String args[]){    }  
    void hello(){System.out.println("Hello");}  
    void welcome(){System.out.println("Welcome to Java 10");}  
}
```

Тут визначено два додаткові методи: `hello` і `welcome`, кожен з яких виводить деякий рядок на консоль. Методи визначаються всередині класу - у цьому випадку всередині класу `Program`, у якому визначено метод `main`.

Методи

Але якщо ми скомпілюємо і запустимо цю програму, то ми нічого не побачимо на консолі. У прикладі вище ми визначили два методи, але ми їх ніде не викликаємо. За замовчуванням у програмі Java виконується тільки метод `main` і весь його вміст. Тому, якщо ми хочемо, щоб інші методи теж виконувалися, їх треба викликати в методі `main`. Виклик методу здійснюється у формі:

назва_методу(аргументи);

Після імені методу вказуються дужки, в яких перераховуються аргументи - значення для параметрів методу.

За допомогою параметрів ми можемо передати в методи різні дані, які будуть використовуватися для обчислень. Наприклад:

```
static void sum(int x, int y){  
    int z = x + y;  
    System.out.println(z);  
}
```

Ця функція приймає два параметри - два числа, додає їх і виводить їхню суму на консоль. А при виклику цього методу в програмі нам необхідно передати на місце параметрів значення, які відповідають типу параметра.

```
sum(10,15);
```

```
sum(20,2);
```

Параметри методів

Оскільки метод `sum` приймає два значення типу `int`, то на місце параметрів треба передати два значення типу `int`. Це можуть бути і числові літерали, і змінні типів даних, які представляють тип `int` або можуть бути автоматично перетворені в тип `int`. Значення, які передаються на місце параметрів, ще називаються аргументами. Значення передаються параметрам за позицією, тобто перший аргумент першому параметру, другий аргумент - другому параметру і так далі.

Параметри змінної довжини

Метод може приймати параметри змінної довжини одного типу. Наприклад, нам треба передати в метод набір чисел і обчислити їхню суму, але ми точно не знаємо, скільки саме чисел буде передано - 3, 4, 5 або більше. Параметри змінної довжини дають змогу вирішити це завдання:

Параметри методів

```
public class Program{  
    public static void main (String args[]){  
        sum(1, 2, 3);           // 6  
        sum(1, 2, 3, 4, 5);     // 15  
        sum();                  // 0  
    }  
    static void sum(int ...nums){  
        int result =0;  
        for(int n: nums)  
            result += n;  
        System.out.println(result);  
    }  
}
```

Трикрапка перед назвою параметра `int ...nums` вказує на те, що він буде необов'язковим і представлятиме масив. Ми можемо передати в метод `sum` одне число, кілька чисел, а можемо взагалі не передавати жодних параметрів. Причому, якщо ми хочемо передати кілька параметрів, то необов'язковий параметр повинен вказуватися в кінці:

Оператор return. Результат методу

Методи можуть повертати деяке значення. Для цього застосовується оператор return.

`return [значення, що повертається];`

Після оператора return вказується значення, що повертається, яке є результатом методу. Це може бути літеральне значення, значення змінної або якогось складного виразу.

Наприклад:

```
public class Program{
    public static void main (String args[]){
        int x = sum(1, 2, 3);
        int y = sum(1, 4, 9);
        System.out.println(x); // 6
        System.out.println(y); // 14
    }
    static int sum(int a, int b, int c){
        return a + b + c;
    }
}
```

Оператор return. Результат методу

У методі як тип значення, що повертається, замість `void` використовується будь-який інший тип. У цьому випадку метод `sum` повертає значення типу `int`, тому цей тип вказується перед назвою методу. Причому якщо як тип, що повертається, для методу визначено будь-який інший, відмінний від **`void`**, то метод обов'язково повинен використовувати оператор **`return`** для повернення значення. При цьому значення, що повертається, завжди повинно мати той самий тип, що значиться у визначенні функції. І якщо функція повертає значення типу `int`, то після оператора **`return`** стоїть цілочисельне значення, яке є об'єктом типу `int`. Як у даному випадку це сума значень параметрів методу. Метод може використовувати кілька викликів оператора **`return`** для повернення різних значень залежно від деяких умов:

Оператор return. Результат методу

```
public class Program{  
    public static void main (String args[]){  
        System.out.println(daytime(7));    // Good morning  
        System.out.println(daytime(13));   // Good after noon  
        System.out.println(daytime(18));   // Good evening  
        System.out.println(daytime(2));    // Good night  
    }  
    static String daytime(int hour){  
        if (hour >24 || hour < 0) return "Invalid data";  
        else if(hour > 21 || hour < 6) return "Good night";  
        else if(hour >= 15) return "Good evening";  
        else if(hour >= 11) return "Good after noon";  
        else return "Good morning";  
    }  
}
```

Тут метод daytime повертає значення типу String, тобто рядок, і залежно від значення параметра hour рядок, що повертається, буде відрізнятися.

Оператор return. Результат методу

Вихід із методу

Оператор return застосовується не тільки для повернення значення з методу, а й для виходу з методу. У подібній якості оператор return застосовується в методах, які нічого не повертають, тобто мають тип void:

```
static void daytime(int hour){  
    if (hour >24 || hour < 0) return;  
    if(hour > 21 || hour < 6) System.out.println("Good night");  
    else if(hour >= 15) System.out.println("Good evening");  
    else if(hour >= 11) System.out.println("Good after noon");  
    else System.out.println("Good morning");  
}
```

Якщо передане в метод daytime значення більше 24 або менше 0, то просто виходимо з методу. Значення, що повертається, після return вказувати в цьому випадку не потрібно.

Перевантаження методів

У програмі ми можемо використовувати методи з одним і тим самим іменем, але з різними типами та/або кількістю параметрів. Такий механізм називається перевантаженням методів (**method overloading**). Наприклад:

```
public class Program{
    public static void main(String[] args) {
        System.out.println(sum(2, 3));           // 5
        System.out.println(sum(4.5, 3.2));       // 7.7
        System.out.println(sum(4, 3, 7));        // 14
    }
    static int sum(int x, int y){
        return x + y;
    }
    static double sum(double x, double y){
        return x + y;
    }
    static int sum(int x, int y, int z){
        return x + y + z;
    }
}
```

Тут визначено три варіанти або три перевантаження методу `sum()`, але під час його виклику залежно від типу та кількості переданих параметрів система вибере саме ту версію, яка найбільше підходить. Варто зазначити, що на перевантаження методів впливають кількість і типи параметрів. Однак відмінність у типі значення, що повертається, для перевантаження не мають жодного значення.

Перевантаження методів

Наприклад, у випадку нижче методи розрізняються за типом значення, що повертається:

```
public class Program{  
    public static void main(String[] args) {  
        System.out.println(sum(2, 3));  
        System.out.println(sum(4, 3));  
    }  
    static int sum(int x, int y){  
        return x + y;  
    }  
    static double sum(int x, int y){  
        return x + y;  
    }  
}
```

Однак перевантаженням це не вважатиметься. Ба більше, така програма некоректна і просто не скомпілюється, оскільки метод з однією і тією ж кількістю і типом параметрів визначено кілька разів.

Рекурсивні функції

Окремо розглянемо рекурсивні функції.

Рекурсія – це техніка здійснення виклику самої функції. Ця техніка дозволяє розбити складні проблеми на простіші, які легше вирішити. Рекурсія може бути трохи складною для розуміння. Найкращий спосіб з'ясувати, як вона працює - це поекспериментувати з нею. Головна відмінність рекурсивних функцій від звичайних методів полягає в тому, що вони рекурсивна функція може викликати саму себе.

Наприклад, розглянемо функцію, яка обчислює факторіал числа:

```
static int factorial(int x){  
    if (x == 1)  
        return 1;  
    return x * factorial(x - 1);  
}
```

Спочатку перевіряється умова: якщо число, що вводиться, не дорівнює 1, то ми множимо це число на результат цієї самої функції, в яку як параметр передається число $x-1$. Тобто відбувається рекурсивний спуск. І так далі, поки не дійдемо того моменту, коли значення параметра не дорівнюватиме одиниці.

Рекурсивні функції

Рекурсивна функція обов'язково повинна мати деякий базовий варіант, який використовує оператор `return` і який поміщається на початку функції. У випадку з факторіалом це `if (x == 1) return 1;`.

І всі рекурсивні виклики мають звертатися до підфункцій, які зрештою сходяться до базового варіанта. Так, у разі передачі у функцію додатного числа під час подальших рекурсивних викликів підфункцій у них передаватиметься щоразу число, менше на одиницю. І врешті-решт ми дійдемо до ситуації, коли число дорівнюватиме 1, і буде використано базовий варіант. Хоча в даному випадку потрібно зазначити, що для визначення факторіалу є більш оптимальні рішення на основі циклів:

```
static int factorial(int x){  
    int result=1;  
    for (int i = 1; i <= x; i++) result *= i;  
    return result;  
}
```

Рекурсивні функції

Ще одним поширеним прикладом рекурсивної функції слугує функція, що обчислює числа Фібоначчі. У теорії n -й член послідовності Фібоначчі визначається за формулою: **$f(n)=f(n-1) + f(n-2)$** , причому **$f(0)=0$** , а **$f(1)=1$** .

```
static int fibonachi(int n){  
    if (n == 0) return 0;  
    if (n == 1) return 1;  
    else return fibonachi(n - 1) + fibonachi(n - 2);  
}
```

Контрольні питання

1. Зазначте основні переваги використання методів у програмуванні та поясніть, чому вони є важливими для покращення якості коду.
2. Розкрийте процедуру створення власного методу у мові програмування Java, включаючи основні кроки для його визначення.
3. Поясніть, як визначати та викликати методи з параметрами у мові програмування Java, і визначте допустимі типи значень, які можна передавати методам.
4. Охарактеризуйте рекурсивні функції в мові програмування Java та подайте приклади використання рекурсії.
5. Розгляньте призначення оператора `return` у методах мови програмування Java та поясніть його роль у поверненні значень.
6. Визначте та поясніть концепцію перевантаження методів в мові програмування Java, і наведіть приклади використання цього механізму.

The END
Lec 3.

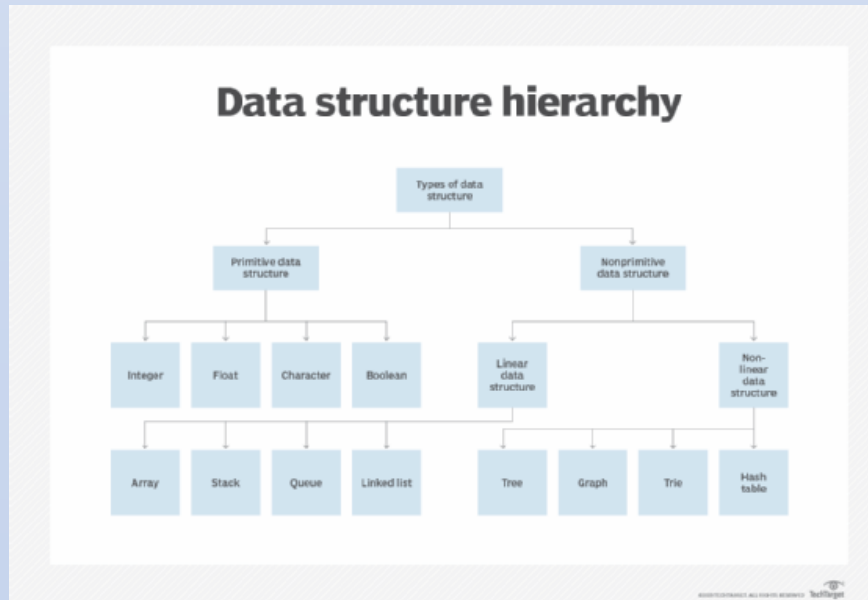
.NET та JAVA #4

Структури даних. Колекції в Java

1. Визначення структур даних
2. Типи колекцій. Інтерфейс Collection
3. Клас ArrayList та інтерфейс List
4. Черги та клас ArrayDeque
5. Клас LinkedList
6. Інтерфейс Set і клас HashSet
7. Інтерфейси SortedSet та NavigableSet. Клас TreeSet
8. Інтерфейси Comparable та Comparator
9. Інтерфейс Map та клас HashMap
10. Ітератори

Визначення структур даних

Структура даних - це спеціалізований формат для організації, обробки, пошуку та зберігання даних. Існує кілька базових і розширених типів структур даних, кожна з яких призначена для впорядкування даних відповідно до певної мети. Структури даних полегшують користувачам доступ до потрібних їм даних і роботу з ними відповідним чином. Найважливіше те, що структури даних визначають організацію інформації так, щоб машини і люди могли краще її розуміти. В інформатиці та комп'ютерному програмуванні структура даних може бути обрана або розроблена для зберігання даних з метою використання їх за допомогою різних алгоритмів. У деяких випадках основні операції алгоритму тісно пов'язані зі структурою даних. Кожна структура даних містить інформацію про значення даних, зв'язки між даними і - в деяких випадках - функції, які можна застосувати до даних.



Визначення структур даних

Існують такі непримітивні структури даних:

Масив. Масив зберігає набір елементів у суміжних комірках пам'яті. Елементи одного типу зберігаються разом, тому позицію кожного елемента можна легко обчислити або знайти за індексом. Масиви можуть бути фіксованої або гнучкої довжини.

Стек. Стек зберігає колекцію елементів у лінійному порядку, в якому застосовуються операції. Цей порядок може бути останнім прийшов, першим пішов (LIFO) або першим прийшов, першим пішов (FIFO).

Черга. Черга зберігає колекцію елементів подібно до стеку, але порядок виконання операцій може бути тільки першим прийшов - першим пішов.

Зв'язаний список. Зв'язаний список зберігає колекцію елементів у лінійному порядку. Кожен елемент, або вузол, зв'язаного списку містить елемент даних, а також посилання на наступний елемент у списку.

Дерево. Дерево зберігає колекцію елементів в абстрактний, ієрархічний спосіб. Кожен вузол пов'язаний з ключовим значенням, а батьківські вузли пов'язані з дочірніми вузлами - або підвузлами.

Граф. Граф зберігає колекцію елементів у нелінійний спосіб. Граф складається зі скінченної множини вузлів, також відомих як вершини, і ліній, що їх з'єднують, також відомих як ребра. Вони корисні для представлення реальних систем, таких як комп'ютерні мережі.

Геш-таблиця, також відома як hashtable, зберігає набір елементів в асоціативному масиві, який відображає ключі до значень. Геш-таблиця використовує геш-функцію для перетворення індексу в масив відер, які містять потрібний елемент даних.

Визначення структур даних

Як обрати структуру даних?

Вибираючи структуру даних для програми або додатку, розробники повинні враховувати відповіді на наступні три питання:

- **Підтримувані операції.** Які функції та операції потрібні програмі?
- **Обчислювальна складність.** Який рівень обчислювальної продуктивності є допустимим? Для швидкості, структура даних, операції якої виконуються за час, лінійний до кількості керованих елементів - використовуючи нотацію Big O Notation: $O(n)$ -- буде швидшою, ніж структура даних, операції якої виконуються за час, пропорційний квадрату кількості елементів, що обробляються -- $O(n^2)$.
- **Елегантність програмування.** Чи є організація структури даних та її функціональний інтерфейс простими у використанні?

	Обчислювальна складність							
	Середнє				Найгірше			
	Індекс	Пошук	Вставка	Видалення	Індекс	Пошук	Вставка	Видалення
ArrayList	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Vector	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$
LinkedList	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Hashtable	n/a	$O(1)$	$O(1)$	$O(1)$	n/a	$O(n)$	$O(n)$	$O(n)$
HashMap	n/a	$O(1)$	$O(1)$	$O(1)$	n/a	$O(n)$	$O(n)$	$O(n)$
LinkedHashMap	n/a	$O(1)$	$O(1)$	$O(1)$	n/a	$O(n)$	$O(n)$	$O(n)$
TreeMap	n/a	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	n/a	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$
HashSet	n/a	$O(1)$	$O(1)$	$O(1)$	n/a	$O(n)$	$O(n)$	$O(n)$
LinkedHashSet	n/a	$O(1)$	$O(1)$	$O(1)$	n/a	$O(n)$	$O(n)$	$O(n)$
TreeSet	n/a	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	n/a	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$

Визначення структур даних

Ось деякі приклади з реального життя:

- **Зв'язані списки** найкраще використовувати, якщо програма керує колекцією елементів, які не потрібно впорядковувати, для додавання або видалення елемента з колекції потрібен постійний час, а збільшення часу пошуку є нормальним явищем;
- **Стеки** найкраще використовувати, якщо програма керує колекцією, яка має підтримувати порядок LIFO;
- **Черги** слід використовувати, якщо програма керує колекцією, яка має підтримувати порядок FIFO;
- **Бінарні дерева** добре підходять для керування колекцією елементів з відношенням "батько-дитина", наприклад, генеалогічним деревом;
- **Бінарні дерева** пошуку підходять для управління відсортованою колекцією, де метою є оптимізація часу, необхідного для пошуку певних елементів у колекції;
- **Графи** найкраще працюють, якщо програма аналізує зв'язки та відносини між колекцією осіб у соціальній мережі.

Типи колекцій. Інтерфейс Collection

Для зберігання наборів даних у Java призначені масиви. Однак їх не завжди зручно використовувати, насамперед тому, що вони мають фіксовану довжину. Цю проблему в Java вирішують колекції. Однак суть не тільки в гнучких за розміром наборах об'єктів, а й у тому, що класи колекцій реалізують різні алгоритми і структури даних, наприклад, такі як стек, черга, дерево і низка інших.

На цій лекції не буде повного розбору або аналізу усього пакету ***java.util.concurrent***. Але ми пройдемося по основам цього пакету для розуміння роботи колекцій.

Java Collection Framework - ієрархія інтерфейсів та їхніх реалізацій, що є частиною JDK і дає змогу розробнику користуватися великою кількістю структур даних із "коробки".

Типи колекцій. Інтерфейс Collection

Вибір певного класу для роботи з колекціями визначає набір методів, які будуть доступні для об'єкта цього класу. Наприклад, якщо використовується список (який визначає інтерфейс List), то існує багатий вибір для його реалізації: **ArrayList**, **LinkedList**, **Vector**, **Stack**. Конкретний вибір реалізації списку позначається на ефективності маніпуляцій з об'єктами списку. Так, **ArrayList** зберігає елементи у вигляді масиву, а отже, доступ і заміна буде виконуватися відносно швидко. Водночас **LinkedList** зберігає елементи у вигляді зв'язного списку, що тягне за собою відносно повільний пошук елементів і швидку операцію додавання/видалення. Класи колекцій розташовуються в пакеті `java.util`, тому перед застосуванням колекцій слід підключити цей пакет.

Рекомендується ознайомитися з описами наступних колекцій з офіційної документації:

- ***java.util.Collection;***
- ***java.util.ArrayList;***
- ***java.util.HashMap;***
- ***java.util.HashSet.***

Важливе також знання основних класів для роботи з колекціями. На особливу увагу при вивченні колекцій заслуговують класи **Comparator**, **Collections**, **Iterator**.

Типи колекцій. Інтерфейс Collection

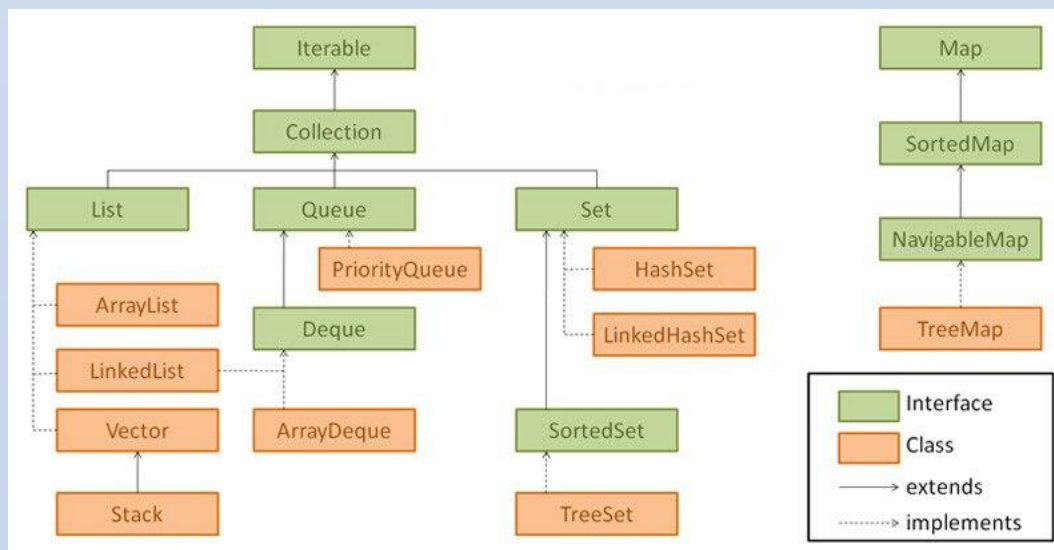
Хоча в Java існує безліч колекцій, але всі вони утворюють струнку і логічну систему. По-перше, в основі всіх колекцій лежить застосування того чи іншого інтерфейсу, який визначає базовий функціонал. Серед цих інтерфейсів можна виділити такі:

- **Collection**: базовий інтерфейс для всіх колекцій та інших інтерфейсів колекцій;
- **Queue**: успадковує інтерфейс Collection і представляє функціонал для структур даних у вигляді черги;
- **Deque**: успадковує інтерфейс Queue і представляє функціонал для двонаправлених черг;
- **List**: успадковує інтерфейс Collection і представляє функціональність простих списків;
- **Set**: також розширює інтерфейс Collection і використовується для зберігання множин унікальних об'єктів;
- **SortedSet**: розширює інтерфейс Set для створення сортованих колекцій;
- **NavigableSet**: розширює інтерфейс SortedSet для створення колекцій, у яких можна здійснювати пошук за відповідністю;
- **Map**: призначений для створення структур даних у вигляді словника, де кожен елемент має певний ключ і значення. На відміну від інших інтерфейсів колекцій не успадковується від інтерфейсу Collection.

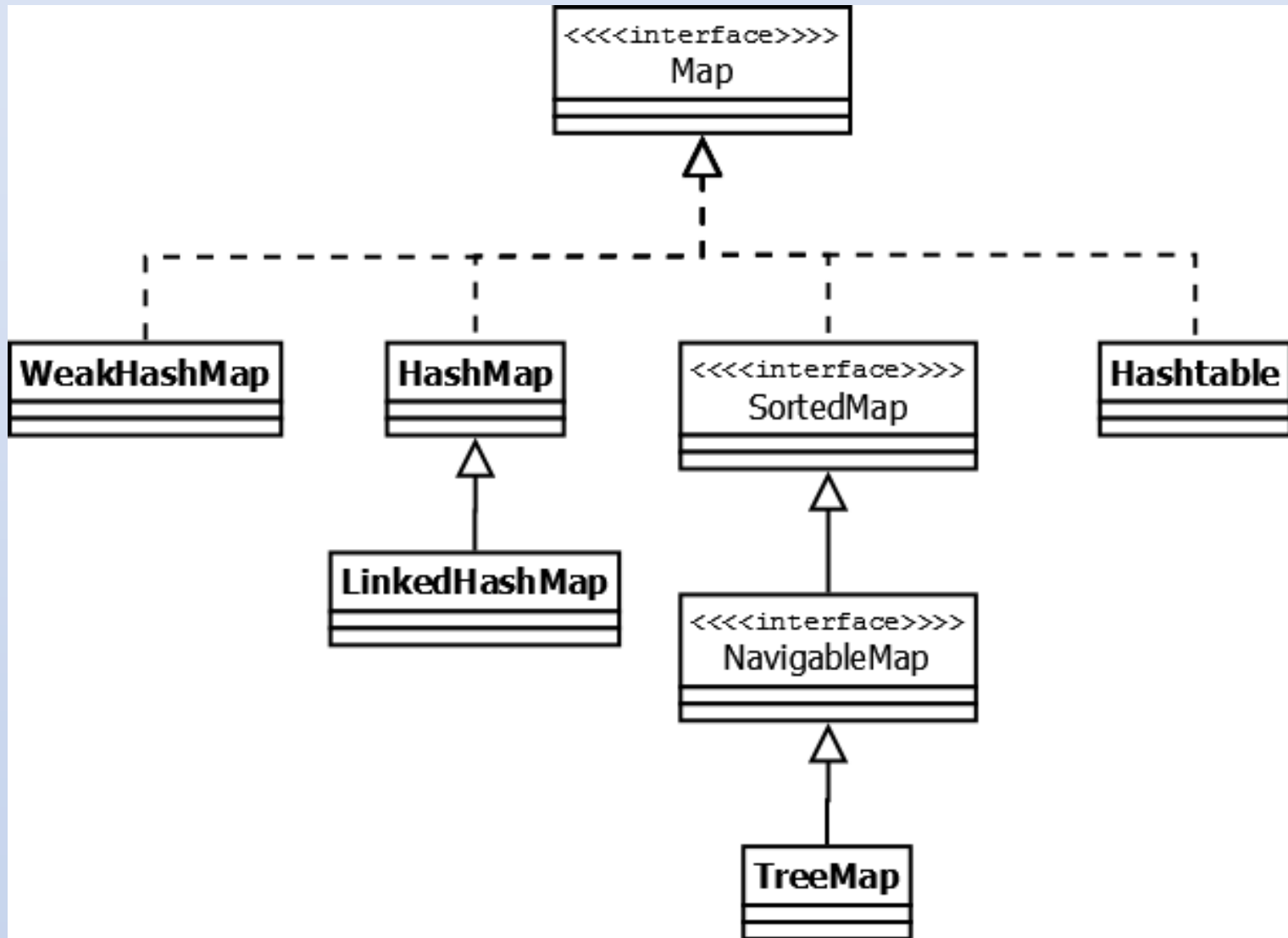
Типи колекцій. Інтерфейс Collection

Ці інтерфейси частково реалізуються абстрактними класами:

- **AbstractCollection**: базовий абстрактний клас для інших колекцій, який застосовує інтерфейс Collection;
- **AbstractList**: розширює клас AbstractCollection і застосовує інтерфейс List, призначений для створення колекцій у вигляді списків;
- **AbstractSet**: розширює клас AbstractCollection і застосовує інтерфейс Set для створення колекцій у вигляді множин;
- **AbstractQueue**: розширює клас AbstractCollection і застосовує інтерфейс Queue, призначений для створення колекцій у вигляді черг і стеків;
- **AbstractSequentialList**: також розширює клас AbstractList і реалізує інтерфейс List. Використовується для створення пов'язаних списків;
- **AbstractMap**: застосовує інтерфейс Map, призначений для створення наборів за типом словника з об'єктами у вигляді пари "ключ-значення".



Типи колекцій. Інтерфейс Collection



Типи колекцій. Інтерфейс Collection

За допомогою застосування інтерфейсів і абстрактних класів у Java реалізується широка палітра класів колекцій - списки, множини, черги, відображення та інші, серед яких можна виділити такі:

- **ArrayList**: простий список об'єктів;
- **LinkedList**: представляє пов'язаний список;
- **ArrayDeque**: клас двоспрямованої черги, в якій ми можемо зробити вставку і видалення як на початку колекції, так і в її кінці;
- **HashSet**: набір об'єктів або хеш-множина, де кожен елемент має ключ - унікальний хеш-код;
- **TreeSet**: набір відсортованих об'єктів у вигляді дерева;
- **LinkedHashSet**: пов'язана хеш-множина;
- **PriorityQueue**: черга пріоритетів;
- **HashMap**: структура даних у вигляді словника, у якому кожен об'єкт має унікальний ключ і деяке значення;
- **TreeMap**: структура даних у вигляді дерева, де кожен елемент має унікальний ключ і деяке значення.

Типи колекцій. Інтерфейс Collection

Інтерфейс Collection є базовим для всіх колекцій, визначаючи основний функціонал:

```
public interface Collection<E> extends Iterable<E>{
```

```
    // визначення методів
```

```
}
```

Інтерфейс Collection є узагальненим і розширює інтерфейс Iterable, тому всі об'єкти колекцій можна перебирати в циклі за типом for-each. Серед методів інтерфейсу Collection можна виділити такі:

- **boolean add (E item)**: додає в колекцію об'єкт item. У разі вдалого додавання повертає true, у разі невдалого – false;
- **boolean addAll (Collection<? extends E> col)**: додає в колекцію всі елементи з колекції col. При вдалому додаванні повертає true, при невдалому – false;
- **void clear ()**: видаляє всі елементи з колекції;
- **boolean contains (Object item)**: повертає true, якщо об'єкт item міститься в колекції, інакше повертає false;
- **boolean isEmpty()**: повертає true, якщо колекція порожня, інакше повертає false;

Типи колекцій. Інтерфейс Collection

- `Iterator<E> iterator ()`: повертає об'єкт `Iterator` для обходу елементів колекції;
- `boolean remove (Object item)`: повертає `true`, якщо об'єкт `item` вдало видалено з колекції, інакше повертається `false`;
- `boolean removeAll (Collection<?> col)`: видаляє всі об'єкти колекції `col` з поточної колекції. Якщо поточна колекція змінилася, повертає `true`, інакше повертається `false`;
- `boolean retainAll (Collection<?> col)`: видаляє всі об'єкти з поточної колекції, крім тих, що містяться в колекції `col`. Якщо поточна колекція після видалення змінилася, повертає `true`, інакше повертається `false`;
- `int size ()`: повертає число елементів у колекції;
- `Object[] toArray ()`: повертає масив, що містить усі елементи колекції.

Усі ці та інші методи, які є в інтерфейсі **Collection**, реалізуються всіма колекціями, тому загалом загальні принципи роботи з колекціями будуть одні й ті самі. Однаковий інтерфейс спрощує розуміння і роботу з різними типами колекцій.

Клас ArrayList та інтерфейс List

Для створення простих списків застосовується інтерфейс List, який розширює функціональність інтерфейсу Collection. Деякі найбільш часто використовувані методи інтерфейсу List:

- **void add(int index, E obj):** додає до списку за індексом index об'єкт obj;
- **boolean addAll(int index, Collection<? extends E> col):** додає до списку за індексом index усі елементи колекції col. Якщо в результаті додавання список був змінений, то повертається true, інакше повертається false;
- **E get(int index):** повертає об'єкт зі списку за індексом index;
- **int indexOf(Object obj):** повертає індекс першого входження об'єкта obj у список. Якщо об'єкт не знайдено, то повертається -1;
- **lastIndexOf(Object obj):** повертає індекс останнього входження об'єкта obj у список. Якщо об'єкт не знайдено, то повертається -1;
- **ListIterator<E> listIterator ():** повертає об'єкт ListIterator для обходу елементів списку;
- **static <E> List<E> of(elements):** створює з набору елементів об'єкт List;
- **E remove(int index):** видаляє об'єкт зі списку за індексом index, повертаючи при цьому видалений об'єкт;
- **E set(int index, E obj):** присвоює значення об'єкта obj елементу, який знаходиться за індексом index;
- **void sort(Comparator<? super E> comp):** сортує список за допомогою компаратора comp;
- **List<E> subList(int start, int end):** отримує набір елементів, що знаходяться у списку між індексами start та end. За замовчуванням у Java є вбудована реалізація цього інтерфейсу - клас ArrayList.

Клас ArrayList та інтерфейс List

Клас **ArrayList** представляє узагальнену колекцію, яка успадковує свою функціональність від класу **AbstractList** і застосовує інтерфейс **List**. Простіше кажучи, ArrayList представляє простий список, аналогічний масиву, за тим винятком, що кількість елементів у ньому не фіксована. ArrayList має такі конструктори:

- **ArrayList()**: створює порожній список;
- **ArrayList(Collection <? extends E> col)**: створює список, до якого додаються всі елементи колекції col;
- **ArrayList (int capacity)**: створює список, який має початкову ємність capacity.

Ємність в **ArrayList** представляє розмір масиву, який буде використовуватися для зберігання об'єктів. Під час додавання елементів фактично відбувається перерозподіл пам'яті - створення нового масиву і копіювання в нього елементів зі старого масиву. Початкове завдання ємності **ArrayList** дає змогу знизити подібні перерозподіли пам'яті, тим самим підвищуючи продуктивність.

Клас ArrayList та інтерфейс List

```
ArrayList<String> people = new ArrayList<String>();  
// додамо до списку низку елементів  
people.add("Tom");  
people.add("Alice");  
people.add("Kate");  
people.add("Sam");  
people.add(1, "Bob"); // додаємо елемент за індексом 1  
System.out.println(people.get(1)); // отримуємо 2-й об'єкт  
people.set(1, "Robert"); // встановлення нового значення для 2-го об'єкта  
  
System.out.printf("ArrayList has %d elements \n", people.size());  
for(String person : people) System.out.println(person);  
// перевіряємо наявність елемента  
  
if(people.contains("Tom")) System.out.println("ArrayList contains Tom");  
//видалимо кілька об'єктів  
// видалення конкретного елемента  
people.remove("Robert");  
  
// видалення за індексом  
people.remove(0);  
  
Object[] peopleArray = people.toArray();  
for(Object person : peopleArray) System.out.println(person);
```


Черги та клас `ArrayDeque`

Хоча ми можемо вільно додавати в об'єкт **`ArrayList`** додаткові об'єкти, на відміну від масиву, проте в реальності **`ArrayList`** використовує для зберігання об'єктів знову ж таки масив. За замовчуванням цей масив призначений для 10 об'єктів. Якщо в процесі програми додається набагато більше, то створюється новий масив, який може вмістити в себе всю кількість. Подібні перерозподіли пам'яті зменшують продуктивність. Тому якщо ми точно знаємо, що у нас список не буде містити більше певної кількості елементів, наприклад, 25, то ми можемо відразу ж явно встановити цю кількість, або в конструкторі: **`ArrayList<String> people = new ArrayList<String>(25);`**, або за допомогою методу **`ensureCapacity`**: **`people.ensureCapacity(25);`**

Черги представляють структуру даних, що працює за принципом **FIFO (first in - first out)**. Тобто чим раніше було додано елемент до колекції, тим раніше він з неї видаляється. Це стандартна модель односпрямованої черги. Однак бувають і двоспрямовані - тобто такі, в яких ми можемо додати елемент не тільки до початку, а й до кінця. І відповідно видалити елемент не тільки з кінця, а й з початку.

Особливістю класів черг є те, що вони реалізують спеціальні інтерфейси **`Queue`** або **`Deque`**.

Черги та клас ArrayDeque

Інтерфейс Queue

Узагальнений інтерфейс **Queue<E>** розширює базовий інтерфейс Collection і визначає поведінку класу як односпрямованої черги. Свою функціональність він розкриває через такі методи:

- **E element():** повертає, але не видаляє, елемент із початку черги. Якщо черга порожня, генерує виняток **NoSuchElementException**;
- **boolean offer(E obj):** додає елемент obj у кінець черги. Якщо елемент вдало доданий, повертає true, інакше – false;
- **E peek():** повертає без видалення елемент з початку черги. Якщо черга порожня, повертає значення null;
- **E poll():** повертає з видаленням елемент з початку черги. Якщо черга порожня, повертає значення null;
- **E remove():** повертає з видаленням елемент з початку черги. Якщо черга порожня, генерує виняток **NoSuchElementException**.

Черги та клас ArrayDeque

Інтерфейс Deque розширює інтерфейс **Queue** і визначає поведінку двоспрямованої черги, яка працює як звичайна односпрямована черга, або як стек, що діє за принципом **LIFO** (останній увійшов - перший вийшов). Інтерфейс Deque визначає такі методи:

- **void addFirst(E obj):** додає елемент на початок черги;
- **void addLast(E obj):** додає елемент obj у кінець черги;
- **E getFirst():** повертає без видалення елемент із голови черги. Якщо черга порожня, генерує виняток NoSuchElementException;
- **E getLast():** повертає без видалення останній елемент черги. Якщо черга порожня, генерує виняток NoSuchElementException;
- **boolean offerFirst(E obj):** додає елемент obj на самий початок черги. Якщо елемент вдало доданий, повертає true, інакше – false;
- **boolean offerLast(E obj):** додає елемент obj у кінець черги. Якщо елемент вдало доданий, повертає true, інакше – false;
- **E peekFirst():** повертає без видалення елемент з початку черги. Якщо черга порожня, повертає значення null.

Черги та клас `ArrayDeque`

- **E peekLast():** повертає без видалення останній елемент черги. Якщо черга порожня, повертає значення `null`;
- **E pollFirst():** повертає з видаленням елемент з початку черги. Якщо черга порожня, повертає значення `null`;
- **E pollLast():** повертає з видаленням останній елемент черги. Якщо черга порожня, повертає значення `null`;
- **E pop():** повертає з видаленням елемент з початку черги. Якщо черга порожня, генерує виняток **NoSuchElementException**;
- **E push(E element):** додає елемент на самий початок черги;
- **E removeFirst():** повертає з видаленням елемент з початку черги. Якщо черга порожня, генерує виняток `NoSuchElementException`;
- **E removeLast():** повертає з видаленням елемент з кінця черги. Якщо черга порожня, генерує виняток **NoSuchElementException**.

Також існують два додаткових методи: **removeFirstOccurrence** та **removeLastOccurrence**.

Черги та клас `ArrayDeque`

Таким чином, наявність методів **pop** і **push** дає змогу класам, що реалізують цей елемент, діяти як стек. Водночас наявний функціонал також дає змогу створювати двоспрямовані черги, що робить класи, які застосовують цей інтерфейс, досить гнучкими.

Клас `ArrayDeque`

У Java черги представлені низкою класів. Один із них - клас **`ArrayDeque<E>`**. Цей клас представляє узагальнену двонаправлену чергу, успадковуючи функціонал від класу **`AbstractCollection`** і застосовуючи інтерфейс **`Deque`**. У класі **`ArrayDeque`** визначено такі конструктори:

- **`ArrayDeque()`**: створює порожню чергу;
- **`ArrayDeque(Collection<? extends E> col)`**: створює чергу, наповнену елементами з колекції `col`;
- **`ArrayDeque(int capacity)`**: створює чергу з початковою ємністю `capacity`. Якщо ми явно не вказуємо початкову ємність, то ємність за замовчуванням дорівнюватиме 16.

Черги та клас ArrayDeque

```
ArrayDeque<String> states = new ArrayDeque<String>();
states.add("Germany");
states.addFirst("France"); // додаємо елемент у самий початок
states.push("Great Britain"); // додаємо елемент у самий початок
states.addLast("Spain"); // додаємо елемент у кінець колекції
states.add("Italy");
// отримуємо перший елемент без видалення
String sFirst = states.getFirst();
System.out.println(sFirst); // Great Britain
// отримуємо останній елемент без видалення
String sLast = states.getLast();
System.out.println(sLast); // Italy
System.out.printf("Queue size: %d \n", states.size()); // 5
// перебір колекції
while(states.peek()!=null) System.out.println(states.pop());
// чергу з об'єктів Person
ArrayDeque<Person> people = new ArrayDeque<Person>();
people.addFirst(new Person("Tom"));
people.addLast(new Person("Nick"));
// перебір без вилучення
for(Person p : people) System.out.println(p.getName());

class Person{
    private String name;
    public Person(String value){name=value;}
    String getName(){return name;}
}
```

Клас LinkedList

Узагальнений клас **LinkedList**<E> представляє структуру даних у вигляді зв'язаного списку. Він успадковується від класу **AbstractSequentialList** і реалізує інтерфейси **List**, **Dequeue** і **Queue**. Тобто він поєднує функціональність роботи зі списком і функціональність черги. Клас **LinkedList** має такі конструктори:

- **LinkedList()**: створює порожній список;
- **LinkedList(Collection<? extends E> col)**: створює список, до якого додає всі елементи колекції **col**.

LinkedList містить усі ті методи, які визначені в інтерфейсах **List**, **Queue**, **Deque**. Деякі з них:

- **addFirst()** / **offerFirst()**: додає елемент на початок списку;
- **addLast()** / **offerLast()**: додає елемент у кінець списку;
- **removeFirst()** / **pollFirst()**: видаляє перший елемент з початку списку;
- **removeLast()** / **pollLast()**: видаляє останній елемент з кінця списку;
- **getFirst()** / **peekFirst()**: отримує перший елемент;
- **getLast()** / **peekLast()**: отримує останній елемент.

Клас LinkedList

```
public static void main(String[] args) {

    LinkedList<String> states = new LinkedList<>();

    states.add("Germany");
    states.add("France");
    states.addLast(e: "Great Britain");
    states.addFirst(e: "Spain");
    states.add(index: 1, element: "Italy");

    System.out.printf("List has %d elements \n", states.size());
    System.out.println(states.get(1));
    states.set(1, "Portugal");
    for(String state : states){ System.out.println(state);}
    if(states.contains("Germany")){
        System.out.println("List contains Germany");
    }
    states.remove(o: "Germany");
    states.removeFirst();
    states.removeLast();

    LinkedList<Person> people = new LinkedList<>();
    people.add(new Person( value: "Mike"));
    people.addFirst(new Person( value: "Tom"));
    people.addLast(new Person( value: "Nick"));
    people.remove(index: 1);

    for(Person p : people){System.out.println(p.getName());}
    Person first = people.getFirst();
    System.out.println(first.getName());
}
```

```
class Person{
    2 usages
    private final String name;
    3 usages
    public Person(String value){name=value;}
    2 usages
    String getName(){return name;}
}
```


Інтерфейс Set і клас HashSet

Інтерфейс **Set** розширює інтерфейс **Collection** і представляє набір унікальних елементів. **Set** не додає нових методів, лише вносить зміни в успадковані. Зокрема, метод `add()` додає елемент у колекцію і повертає `true`, якщо в колекції ще немає такого елемента.

Узагальнений клас `HashSet` представляє хеш-таблицю. Він успадковує свій функціонал від класу **AbstractSet**, а також реалізує інтерфейс **Set**.

Хеш-таблиця представляє таку структуру даних, у якій усі об'єкти мають унікальний ключ або хеш-код. Цей ключ дає змогу унікально ідентифікувати об'єкт у таблиці.

Для створення об'єкта **HashSet** можна скористатися одним із таких конструкторів:

- **HashSet()**: створює порожній список;
- **HashSet(Collection<? extends E> col)**: створює хеш-таблицю, в яку додає всі елементи колекції `col`;
- **HashSet(int capacity)**: параметр `capacity` вказує початкову ємність таблиці, яка за замовчуванням дорівнює 16;
- **HashSet(int capacity, float koef)**: параметр `koef` або коефіцієнт заповнення, значення якого має бути в межах від 0.0 до 1.0, вказує, наскільки має бути заповнена ємність об'єктами, перш ніж відбудеться її розширення. Наприклад, коефіцієнт 0.75 вказує, що при заповненні ємності на 3/4 відбудеться її розширення.

Клас `HashSet` не додає нових методів, реалізуючи лише ті, що оголошені в батьківських класах і застосовуваних інтерфейсах:

Інтерфейс Set і клас HashSet

```
public static void main(String[] args) {

    HashSet<String> states = new HashSet<>();
    states.add("Germany");
    states.add("France");
    states.add("Italy");
    boolean isAdded = states.add("Germany");
    System.out.println(isAdded);
    System.out.printf("Set contains %d elements \n", states.size());
    for(String state : states){ System.out.println(state); }
    states.remove("Germany");
    HashSet<Person> people = new HashSet<Person>();
    people.add(new Person("Mike"));
    people.add(new Person("Tom"));
    people.add(new Person("Nick"));
    for(Person p : people){ System.out.println(p.getName());}

}
```

```
class Person{
    2 usages
    private final String name;
    3 usages
    public Person(String value){name=value;}
    1 usage
    String getName(){return name;}
}
```

Інтерфейси SortedSet та NavigableSet. Клас TreeSet

SortedSet

Інтерфейс SortedSet призначений для створення колекцій, які зберігають елементи у відсортованому вигляді (сортування за зростанням). SortedSet розширює інтерфейс Set, тому така колекція знову ж таки зберігає тільки унікальні значення. SortedSet надає такі методи:

- **E first():** повертає перший елемент набору;
- **E last():** повертає останній елемент набору;
- **SortedSet<E> headSet(E end):** повертає об'єкт SortedSet, який містить усі елементи первинного набору до елемента end;
- **SortedSet<E> subSet(E start, E end):** повертає об'єкт SortedSet, який містить усі елементи первинного набору між елементами start і end;
- **SortedSet<E> tailSet(E start):** повертає об'єкт SortedSet, який містить усі елементи первинного набору, починаючи з елемента start.

Інтерфейси SortedSet та NavigableSet. Клас TreeSet

Інтерфейс **NavigableSet** розширює інтерфейс SortedSet і дає змогу витягувати елементи на підставі їхніх значень. NavigableSet визначає такі методи:

- **E ceiling(E obj):** шукає в наборі найменший елемент e , який більший за obj ($e > obj$). Якщо такий елемент знайдено, то його повертають як результат. Інакше повертається null;
- **E floor(E obj):** шукає в наборі найбільший елемент e , який менший за елемент obj ($e < obj$). Якщо такий елемент знайдено, то він повертається як результат. Інакше повертається null;
- **E higher(E obj):** шукає в наборі найменший елемент e , який більший за елемент obj ($e > obj$). Якщо такий елемент знайдено, то він повертається як результат. Інакше повертається null;
- **E lower(E obj):** шукає в наборі найбільший елемент e , який менший за елемент obj ($e < obj$). Якщо такий елемент знайдено, то він повертається як результат. Інакше повертається null;

Інтерфейси SortedSet та NavigableSet. Клас TreeSet

- **E pollFirst():** повертає перший елемент і видаляє його з набору;
- **E pollLast():** повертає останній елемент і видаляє його з набору;
- **NavigableSet<E> descendingSet():** повертає об'єкт **NavigableSet**, який містить усі елементи первинного набору **NavigableSet** у зворотному порядку;
- **NavigableSet<E> headSet(E upperBound, boolean incl):** повертає об'єкт **NavigableSet**, який містить усі елементи первинного набору **NavigableSet** до **upperBound**. Параметр **incl** при значенні **true**, дозволяє включити у вихідний набір елемент **upperBound**;
- **NavigableSet<E> tailSet(E lowerBound, boolean incl):** повертає об'єкт **NavigableSet**, який містить усі елементи первинного набору **NavigableSet**, починаючи з **lowerBound**. Параметр **incl** при значенні **true**, дозволяє включити у вихідний набір елемент **lowerBound**;
- **NavigableSet<E> subSet(E lowerBound, boolean lowerIncl, E upperBound, boolean highIncl):** повертає об'єкт **NavigableSet**, який містить усі елементи первинного набору **NavigableSet** від **lowerBound** до **upperBound**.

Інтерфейси SortedSet та NavigableSet. Клас TreeSet

Узагальнений клас **TreeSet**<E> представляє структуру даних у вигляді дерева, в якому всі об'єкти зберігаються у відсортованому вигляді за зростанням. TreeSet є спадкоємцем класу AbstractSet і реалізує інтерфейс NavigableSet, а отже, й інтерфейс SortedSet. У класі TreeSet визначено такі конструктори:

- **TreeSet()**: створює порожнє дерево;
- **TreeSet(Collection<? extends E> col)**: створює дерево, в яке додає всі елементи колекції col;
- **TreeSet(SortedSet <E> set)**: створює дерево, в яке додає всі елементи сортованого набору set;
- **TreeSet(Comparator<? super E> comparator)**: створює порожнє дерево, в яке всі елементи, що додаються, згодом будуть відсортовані компаратором.

TreeSet підтримує всі стандартні методи для вставки та видалення елементів.

```
public static void main(String[] args) {  
  
    TreeSet<String> states = new TreeSet<>();  
    states.add("Germany");  
    states.add("France");  
    states.add("Italy");  
    states.add("Great Britain");  
    System.out.printf("TreeSet contains %d elements \n", states.size());  
    states.remove("Germany");  
    for(String state : states){  
        System.out.println(state);  
    }  
}
```

Інтерфейси SortedSet та NavigableSet. Клас TreeSet

Оскільки TreeSet реалізує інтерфейс NavigableSet, а через нього і SortedSet, то ми можемо застосувати до структури дерева різні методи:

```
public static void main(String[] args) {

    TreeSet<String> states = new TreeSet<>();
    states.add("Germany");
    states.add("France");
    states.add("Italy");
    states.add("Spain");
    states.add("Great Britain");
    System.out.println(states.first()); // отримаємо перший - самий менший елемент
    System.out.println(states.last()); // отримаємо останній - самий найбільший елемент
    // отримаємо піднабір від одного елемента до іншого
    SortedSet<String> set = states.subSet("Germany", "Italy");
    System.out.println(set);
    // елемент із набору, який більший за поточний
    String greater = states.higher(e: "Germany");
    // елемент із набору, який менший за поточний
    String lower = states.lower(e: "Germany");
    // повертаємо набір у зворотному порядку
    NavigableSet<String> navSet = states.descendingSet();
    // повертаємо набір у якому всі елементи менші за поточний
    SortedSet<String> setLower=states.headSet( toElement: "Germany");
    // повертаємо набір у якому всі елементи більші за поточний
    SortedSet<String> setGreater=states.tailSet( fromElement: "Germany");
    System.out.println(navSet);
    System.out.println(setLower);
    System.out.println(setGreater);
}
```

Інтерфейси Comparable та Comparator

До цього було розглянуто роботу колекції **TreeSet**, типізованої об'єктами **String**. При додаванні нових елементів об'єкт **TreeSet** автоматично проводить сортування, поміщаючи новий об'єкт на правильне для нього місце. Однак із рядками все зрозуміло. А що якби ми використовували не рядки, а свої класи, наприклад, наступний клас Person:

```
class Person{
    private String name;
    Person(String name){ this.name=name;}
    String getName(){return name;}
}
```

Об'єкт **TreeSet** ми не зможемо типізувати даним класом, оскільки в разі додавання об'єктів **TreeSet** не знатиме, як їх порівнювати, і наступний шматок коду не працюватиме:

```
TreeSet<Person> people = new TreeSet<Person>();
people.add(new Person("Tom"));
```


Інтерфейси Comparable та Comparator

Під час виконання цього коду ми зіткнемося з помилкою, яка скаже, що об'єкт `Person` не може бути перетворений до типу ***java.lang.Comparable***. Для того, щоб об'єкти `Person` можна було порівняти і сортувати, вони повинні застосовувати інтерфейс **`Comparable<E>`**. При застосуванні інтерфейсу він типізується поточним класом. Застосуємо його до класу `Person`:

```
class Person implements Comparable<Person>{
    private String name;
    Person(String name){
        this.name = name;
    }

    String getName(){return name;}

    public int compareTo(Person p){
        return name.compareTo(p.getName());
    }
}
```

Інтерфейси Comparable та Comparator

Інтерфейс **Comparable** містить один єдиний метод `int compareTo(E item)`, який порівнює поточний об'єкт з об'єктом, переданим як параметра. Якщо цей метод повертає від'ємне число, то поточний об'єкт буде розташовуватися перед тим, який передається через параметр. Якщо метод поверне позитивне число, то, навпаки, після другого об'єкта. Якщо метод поверне нуль, значить, обидва об'єкти рівні.

У цьому випадку ми не повертаємо явно ніякого числа, а покладаємося на вбудований механізм порівняння, який є у класу `String`. Але ми також можемо визначити і свою логіку, наприклад, порівнювати за довжиною імені:

```
public int compareTo(Person p){  
    return name.length() - p.getName().length();  
}
```

Тепер ми можемо типізувати `TreeSet` типом `Person` і додавати в дерево відповідні об'єкти:

```
TreeSet<Person> people = new TreeSet<Person>();  
people.add(new Person("Tom"));
```

Інтерфейси Comparable та Comparator

Інтерфейс Comparator

Однак перед нами може виникнути проблема, що якщо розробник не реалізував у своєму класі, який ми хочемо використовувати, інтерфейс **Comparable**, або реалізував, але нас не влаштовує його функціональність, і ми хочемо її перевизначити? На цей випадок є ще більш гнучкий спосіб, що передбачає застосування інтерфейсу **Comparator<E>**. Інтерфейс **Comparator** містить низку методів, ключовим з яких є метод `compare()`:

```
public interface Comparator<E> {  
    int compare(T a, T b);  
    // інші методи  
}
```

Метод **compare** також повертає числове значення - якщо воно від'ємне, то об'єкт `a` передує об'єкту `b`, інакше - навпаки. А якщо метод повертає нуль, то об'єкти рівні. Для застосування інтерфейсу нам спочатку треба створити клас компаратора, який реалізує цей інтерфейс:

Інтерфейси Comparable та Comparator

```
class PersonComparator implements Comparator<Person>{  
    public int compare(Person a, Person b){  
        return a.getName().compareTo(b.getName());  
    }  
}
```

Тут знову ж таки проводимо порівняння за рядками. Тепер використовуємо клас компаратора для створення об'єкта **TreeSet**:

```
PersonComparator pcomp = new PersonComparator();  
TreeSet<Person> people = new TreeSet<Person>(pcomp);  
people.add(new Person("Tom"));  
people.add(new Person("Nick"));  
people.add(new Person("Alice"));  
people.add(new Person("Bill"));  
for(Person p : people)  
    System.out.println(p.getName());
```

Для створення **TreeSet** тут використовується одна з версій конструктора, яка як параметр приймає компаратор. Тепер незалежно від того, чи реалізовано в класі Person інтерфейс Comparable, логіка порівняння і сортування буде використовуватися та, яка визначена в класі компаратора.

Інтерфейс Map та клас HashMap

Інтерфейс **Map**<K, V> представляє відображення або інакше кажучи словник, де кожен елемент представляє пару "ключ-значення". При цьому всі ключі унікальні в межах об'єкта Map. Такі колекції полегшують пошук елемента, якщо нам відомий ключ - унікальний ідентифікатор об'єкта.

Слід зазначити, що на відміну від інших інтерфейсів, які представляють колекції, інтерфейс Map НЕ розширює інтерфейс **Collection**. Серед методів інтерфейсу Map можна виділити такі:

- **void clear():** очищає колекцію;
- **boolean containsKey(Object k):** повертає true, якщо колекція містить ключ k;
- **boolean containsValue(Object v):** повертає true, якщо колекція містить значення v;
- **Set<Map.Entry<K, V>> entrySet():** повертає набір елементів колекції. Усі елементи представляють об'єкт Map.Entry;
- **boolean equals(Object obj):** повертає true, якщо колекція ідентична колекції, що передається через параметр obj.

Інтерфейс Map та клас HashMap

- **boolean isEmpty:** повертає true, якщо колекція порожня;
- **V get(Object k):** повертає значення об'єкта, ключ якого дорівнює k. Якщо такого елемента не виявиться, то повертається значення null;
- **V getOrDefault(Object k, V defaultValue):** повертає значення об'єкта, ключ якого дорівнює k. Якщо такого елемента не виявиться, то повертається значення defaultValue;
- **V put(K k, V v):** поміщає в колекцію новий об'єкт із ключем k і значенням v. Якщо в колекції вже є об'єкт з подібним ключем, то він перезаписується. Після додавання повертає попереднє значення для ключа k, якщо він уже був у колекції. Якщо ж ключа ще не було в колекції, то повертається значення null;
- **V putIfAbsent(K k, V v):** поміщає в колекцію новий об'єкт з ключем k і значенням v, якщо в колекції ще немає елемента з подібним ключем;
- **Set<K> keySet():** повертає набір усіх ключів відображення;
- **Collection<V> values():** повертає набір усіх значень відображення
- **void putAll(Map<? extends K, ? extends V> map):** додає в колекцію всі об'єкти з відображення map;
- **V remove(Object k):** видаляє об'єкт із ключем k **int size():** повертає кількість елементів колекції.

Інтерфейс Map та клас HashMap

Щоб покласти об'єкт у колекцію, використовується метод `put`, а щоб отримати за ключем - метод `get`. Реалізація інтерфейсу `Map` також дає змогу отримати набори як ключів, так і значень. А метод **`entrySet()`** повертає набір усіх елементів у вигляді об'єктів **`Map.Entry<K, V>`**.

Узагальнений інтерфейс `Map.Entry<K, V>` представляє об'єкт із ключем типу `K` і значенням типу `V` і визначає такі методи:

- **`boolean equals(Object obj)`**: повертає `true`, якщо об'єкт `obj`, що представляє інтерфейс `Map.Entry`, ідентичний поточному;
- **`K getKey()`**: повертає ключ об'єкта відображення;
- **`V getValue()`**: повертає значення об'єкта відображення;
- **`V setValue(V v)`**: встановлює для поточного об'єкта значення `v`;
- **`int hashCode()`**: повертає хеш-код цього об'єкта.

Під час перебору об'єктів відображення ми будемо оперувати цими методами для роботи з ключами і значеннями об'єктів.

Самостійно розберіть тему інтерфейсів `SortedMap`, `NavigableMap` та класу `TreeMap`

Інтерфейс Map та клас HashMap

```
public static void main(String[] args) {  
  
    Map<Integer, String> states = new HashMap<>();  
    states.put(1, "Germany");  
    states.put(2, "Spain");  
    states.put(4, "France");  
    states.put(3, "Italy");  
    String first = states.get(2);  
    System.out.println(first);  
    Set<Integer> keys = states.keySet();  
    Collection<String> values = states.values();  
    states.replace(1, "Poland");  
    states.remove(key: 2);  
    for (Map.Entry<Integer, String> item : states.entrySet()) {  
        System.out.printf("Key: %d Value: %s \n", item.getKey(), item.getValue());  
    }  
    Map<String, Person> people = new HashMap<>();  
    people.put("1240i54", new Person(value: "Tom"));  
    people.put("1564i55", new Person(value: "Bill"));  
    people.put("4540i56", new Person(value: "Nick"));  
  
    for (Map.Entry<String, Person> item : people.entrySet()) {  
        System.out.printf("Key: %s Value: %s \n", item.getKey(), item.getValue().getName());  
    }  
}
```


Ітератори

Одним із ключових методів інтерфейсу **Collection** є метод **Iterator<E> iterator()**. Він повертає **ітератор** - тобто об'єкт, що реалізує інтерфейс **Iterator**. Інтерфейс **Iterator** має таке визначення:

```
public interface Iterator <E>{  
    E next();  
    boolean hasNext();  
    void remove();  
}
```

Реалізація інтерфейсу передбачає, що за допомогою виклику методу `next()` можна отримати наступний елемент. За допомогою методу `hasNext()` можна дізнатися, чи є наступний елемент, і чи не досягнуто кінця колекції. І якщо елементи ще є, то `hasNext()` поверне значення `true`. Метод `hasNext()` слід викликати перед методом `next()`, оскільки в разі досягнення кінця колекції метод `next()` викидає виняток `NoSuchElementException`. І метод `remove()` видаляє поточний елемент, який був отриманий останнім викликом `next()`. Використовуємо ітератор для перебору колекції `ArrayList`:

Ітератори

```
public static void main(String[] args) {  
  
    ArrayList<String> states = new ArrayList<>();  
    states.add("Germany");  
    states.add("France");  
    states.add("Italy");  
    states.add("Spain");  
  
    Iterator<String> iter = states.iterator();  
    while(iter.hasNext()){  
        System.out.println(iter.next());  
    }  
}
```

Інтерфейс Iterator надає обмежений функціонал. Набагато більший набір методів надає інший ітератор - інтерфейс ListIterator. Цей ітератор використовується класами, що реалізують інтерфейс List, тобто класами LinkedList, ArrayList тощо.

Ітератори

Інтерфейс **ListIterator** розширює інтерфейс **Iterator** і визначає низку додаткових методів:

- **void add(E obj):** вставляє об'єкт obj перед елементом, який має бути повернутий наступним викликом next();
- **boolean hasNext():** повертає true, якщо в колекції є наступний елемент, інакше повертає false;
- **boolean hasPrevious():** повертає true, якщо в колекції є попередній елемент, інакше повертає false;
- **E next():** повертає поточний елемент і переходить до наступного, якщо такого немає, то генерується виняток **NoSuchElementException**;
- **E previous():** повертає поточний елемент і переходить до попереднього, якщо такого немає, то генерується виняток **NoSuchElementException**;
- **int nextIndex():** повертає індекс наступного елемента. Якщо такого немає, то повертається розмір списку;
- **int previousIndex():** повертає індекс попереднього елемента. Якщо такого немає, то повертається число -1;
- **void remove():** видаляє поточний елемент зі списку. Таким чином, цей метод має бути викликаний після методів next() або previous(), інакше буде згенеровано виняток **IllegalStateException**;
- **void set(E obj):** присвоює поточному елементу, обраному викликом методів next() або previous(), посилання на об'єкт obj.

Ітератори

```
public static void main(String[] args) {

    ArrayList<String> states = new ArrayList<>();
    states.add("Germany");
    states.add("France");
    states.add("Italy");
    states.add("Spain");

    ListIterator<String> listIter = states.listIterator();
    while(listIter.hasNext()){
        System.out.println(listIter.next());
    }
    // зараз поточний елемент - Іспанія
    // змінимо значення цього елемента
    listIter.set("Португалія");
    // пройдемося по елементах у зворотному порядку
    while(listIter.hasPrevious()){
        System.out.println(listIter.previous());
    }
}
```

Контрольні питання

1. Поясніть концепцію структур даних в мові програмування Java та визначте їх відмінності від звичайних змінних;
2. Розгляньте основні типи колекцій, що доступні у мові програмування Java, та визначте їхню роль у роботі з даними;
3. Визначте ключові операції, які можна виконати з колекціями в Java, такі як додавання, видалення та пошук елементів;
4. Охарактеризуйте приклади завдань, для яких ефективно використовуються структури даних та колекції в мові програмування Java, і поясніть, як це сприяє покращенню ефективності програм;
5. Розгляньте основні методи доступу до даних в різних колекціях, таких як ArrayList або HashMap, та поясніть, як їх правильно використовувати для отримання та зміни даних.

The END
Lec 4.

.NET та JAVA #5

Об'єктно-орієнтоване програмування на мові Java

1. Парадигми програмування
2. Класи та об'єкти
3. Конструктори
4. Пакети
5. Модифікатори доступу. Інкапсуляція
6. Статичні члени та модифікатор static
7. Об'єкти як параметри методів
8. Внутрішні та вкладені класи
9. Успадкування
10. Ієрархія успадкування та перетворення типів
11. Клас Object і його методи
12. Поліморфізм
13. Додаткова інформація

Парадигми програмування

Варто почати, що існують такі основні **парадигми програмування**:

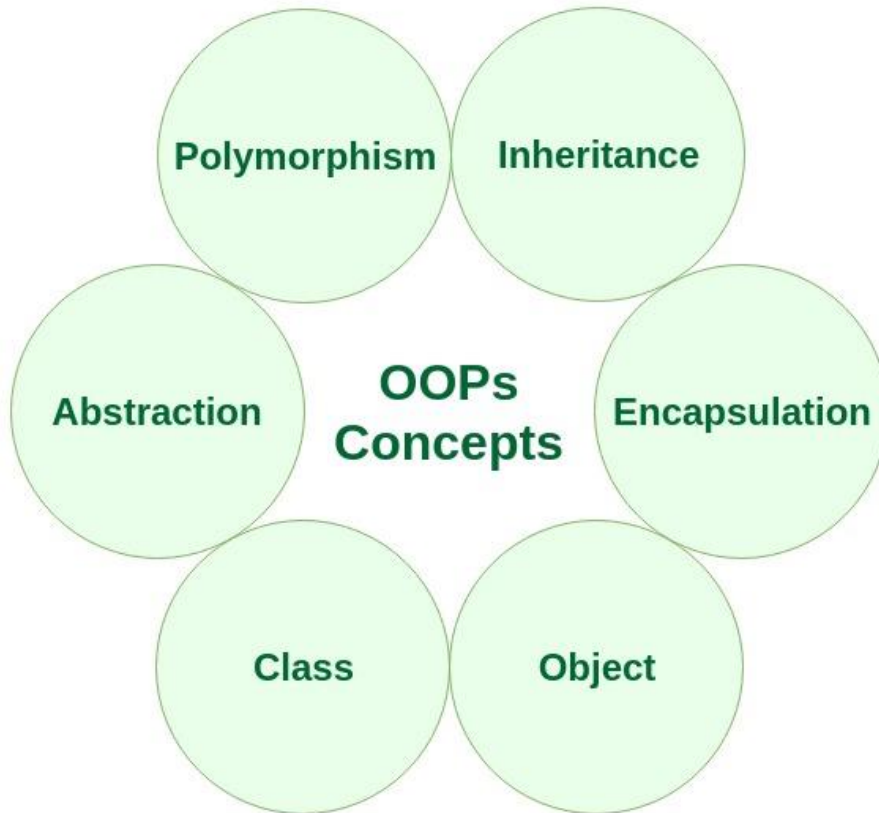
Імперативне програмування - це парадигма програмування, в якій програма являє собою набір інструкцій, що виконуються по порядку. У цій парадигмі основними конструкціями є команди, цикли та умовні оператори. Програміст вказує, які операції мають бути виконані і в якому порядку.

Приклади мов, заснованих на імперативній парадигмі: C, C++, Java, Python, Ruby.

Функціональне програмування - це парадигма програмування, у якій програма являє собою набір функцій, які можуть бути застосовані до даних. У цій парадигмі функції є основними конструкціями, а стан програми незмінний.

Приклади мов, заснованих на функціональній парадигмі: Haskell, Lisp, ML.

Парадигми програмування



ООП (об'єктно-орієнтоване програмування) - це парадигма програмування, що базується на концепції об'єктів, які можуть містити дані (атрибути) і методи, що можуть працювати з цими даними. Основна мета ООП - зв'язати разом дані та функції, що оперують ними, таким чином, щоб жодна інша частина коду не могла отримати доступ до цих даних, окрім цієї функції.

Головна перевага ООП – це **концепції** ООП!

Класи та об'єкти

Java є об'єктно-орієнтованою мовою, тому такі поняття як "**клас**" і "**об'єкт**" відіграють у ній ключову роль. Будь-яку програму на Java можна уявити як набір об'єктів, що взаємодіють між собою.

Шаблоном або описом об'єкта є **клас**, а **об'єкт** являє собою екземпляр цього класу. Можна ще провести таку аналогію. У нас у всіх є деяке уявлення про людину - наявність двох рук, двох ніг, голови, тулуба тощо. Є деякий шаблон - цей шаблон можна назвати класом. Реально ж існуюча людина (фактично екземпляр даного класу) є об'єктом цього класу.

Клас визначається за допомогою ключового слова **class**:

```
class Person {}
```

У цьому випадку клас називається **Person**. Після назви класу йдуть фігурні дужки, між якими поміщається тіло класу - тобто його поля і методи. Будь-який об'єкт може мати дві основні характеристики: стан - деякі дані, які зберігає об'єкт, і поведінка - дії, які може здійснювати об'єкт. Для зберігання стану об'єкта в класі застосовуються поля або змінні класу. Для визначення поведінки об'єкта в класі застосовуються методи. Наприклад, клас **Person**, який представляє людину, міг би мати таке визначення:

Класи та об'єкти

```
class Person{
    String name;        // ім'я
    int age;            // вік
    void displayInfo(){
        System.out.printf("Name: %s \tAge: %d\n", name, age);
    }
}
```

У класі Person визначено два поля: name представляє ім'я людини, а age - її вік. І також визначено метод displayInfo, який нічого не повертає і просто виводить ці дані на консоль.

Для того аби використати цей клас, ми можемо визначити в ньому метод main – який є «початком» нашої програми та створити об'єкт: **Person tom**

Класи та об'єкти

```
public class Program{  
    public static void main(String[] args) {  
        Person tom;  
    }  
}
```

Увага! Зверніть увагу на модифікатор доступу класу `Program`, тільки один клас в файлі може бути `public`, а сам файл коду має мати таку ж саму назву.

Клас представляє новий тип, тому ми можемо визначати змінні, які представляють даний тип. Так, тут у методі `main` визначено змінну `tom`, яка представляє клас `Person`. Але поки що ця змінна не вказує ні на який об'єкт і за замовчуванням вона має значення `null`. За великим рахунком ми її поки не можемо використовувати, тому спочатку необхідно створити об'єкт класу `Person`.

Конструктори

Крім звичайних методів класи можуть визначати спеціальні методи, які називаються **конструкторами**.

Конструктори викликаються при створенні нового об'єкта даного класу. Конструктори виконують ініціалізацію об'єкта. Якщо в класі не визначено жодного конструктора, то для цього класу автоматично створюється конструктор без параметрів.

Раніше визначений клас **Person** не має жодних конструкторів. Тому для нього автоматично створюється конструктор за замовчуванням, який ми можемо використовувати для створення об'єкта Person.

Конструктори

```
public class Program{  
    public static void main(String[] args) {  
        Person tom = new Person(); // создание объекта  
        tom.displayInfo();  
        // змінюємо ім'я та вік  
        tom.name = "Tom";  
        tom.age = 34;  
        tom.displayInfo();  
    }  
}
```

Для створення об'єкта **Person** використовується вираз `new Person()`. Оператор `new` виділяє пам'ять для об'єкта `Person`. І потім викликається конструктор за замовчуванням, який не приймає жодних параметрів. У підсумку після виконання цього виразу в пам'яті буде виділено ділянку, де зберігатимуться всі дані об'єкта `Person`. А змінна `tom` отримає посилання на створений об'єкт.

Якщо конструктор не ініціалізує значення змінних об'єкта, то вони отримують значення за замовчуванням. Для змінних числових типів це число 0, а для типу `string` і класів - це значення `null` (тобто фактично відсутність значення).

Конструктори

Після створення об'єкта ми можемо звернутися до змінних об'єкта **Person** через змінну `tom` і встановити або отримати їхні значення, наприклад, **`tom.name`** = "Tom".

Якщо необхідно, щоб при створенні об'єкта проводилася якась логіка, наприклад, щоб поля класу отримували якісь певні значення, то можна визначити в класі свої конструктори, наприклад:

```
class Person{
    String name;
    int age;
    Person(){
        name = "Undefined";
        age = 18;
    }
    Person(String n){
        name = n;
        age = 18;
    }
    Person(String n, int a){
        name = n;
        age = a;
    }
    void displayInfo(){
        System.out.printf("Name: %s \tAge: %d\n", name, age);
    }
}
```

Конструктори

У третьому конструкторі параметри називаються так само, як і поля класу. І щоб розмежувати поля і параметри, застосовується ключове слово `this`:

```
this.name = name;
```

Так, у цьому випадку вказуємо, що значення параметра `name` присвоюється полю `name`. Крім того, у нас три конструктори, які виконують ідентичні дії: встановлюють поля `name` і `age`. Щоб уникнути повторів, за допомогою `this` можна викликати один із конструкторів класу і передати для його параметрів необхідні значення:

```
Person(String name){  
    this(name, 18);  
}
```


Конструктори

Ініціалізатори

Крім конструктора початкову ініціалізацію об'єкта цілком можна було проводити за допомогою ініціалізатора об'єкта. Ініціалізатор виконується до будь-якого конструктора.

Тобто в ініціалізатор ми можемо помістити код, спільний для всіх конструкторів:

```
class Person{  
  
    String name;  
    int age;  
    /*початок блоку ініціалізатора*/  
    {  
        name = "Undefined";  
        age = 18;  
    }  
    /*кінець блоку ініціалізатора*/  
    Person(){  
  
    }  
}
```

Пакети

Як правило, в Java класи об'єднуються в **пакети**. Пакети дозволяють організувати класи логічно в набори. За замовчуванням java вже має низку вбудованих пакетів, наприклад, **java.lang**, **java.util**, **java.io** тощо. Крім того, пакети можуть мати вкладені пакети.

Організація класів у вигляді пакетів дає змогу уникнути конфлікту імен між класами. Адже нерідкі ситуації, коли розробники називають свої класи однаковиими іменами.

Належність до пакета дає змогу гарантувати однозначність імен. Щоб вказати, що клас належить певному пакету, треба використовувати директиву `package`, після якої вказується ім'я пакета:
`package назва_пакету;`

Пакети

Як правило, назви пакетів відповідають фізичній структурі проекту, тобто організації каталогів, у яких знаходяться файли з вихідним кодом. А шлях до файлів усередині проекту відповідає назві пакета цих файлів. Наприклад, якщо класи належать пакету **mypack**, то ці класи поміщаються в проєкті в папку **mypack**. Класи необов'язково визначати в пакети.

Якщо для класу пакет не визначено, то вважається, що цей клас знаходиться в пакеті за замовчуванням, який не має імені. Наприклад, створимо в папці для вихідних файлів каталог `study`. У ньому створимо файл `Program.java` з таким кодом:

```
package study;
```

```
public class Program{}  
class Person{}
```

Директива `package study` на початку файлу вказує, що класи `Program` і `Person`, які тут визначені, належать пакету `study`.

Пакети

Імпорт пакетів і класів

Якщо нам треба використовувати класи з інших пакетів, то нам треба підключити ці пакети та класи. Виняток становлять класи з пакета **java.lang** (наприклад, **String**), які підключаються в програму автоматично.

Наприклад, знайомий за минулими темами клас **Scanner** міститься в пакеті **java.util**, тому ми можемо отримати до нього доступ у такий спосіб:

```
java.util.Scanner in = new java.util.Scanner(System.in);
```

Тобто ми вказуємо повний шлях до файлу в пакеті під час створення його об'єкта. Однак таке нагромадження імен пакетів не завжди зручне, і як альтернативу ми можемо імпортувати пакети і класи в проєкт за допомогою директиви **import**, яка вказується після директиви **package**:

```
import java.util.Scanner;
```

Пакети

Статичний імпорт

У java є також особлива форма імпорту - статичний імпорт. Для цього разом із директивою `import` використовується модифікатор `static`:

```
package study;
import static java.lang.Math.*;

public class Program {
    public static void main(String[] args) {
        double result = sqrt(20);
        System.out.println(result);
    }
}
```

Тут відбувається статичний імпорт класів `System` і `Math`.

Ці класи мають статичні методи. Завдяки операції статичного імпорту ми можемо використовувати ці методи без назви класу. Наприклад, писати не `Math.sqrt(20)`, а `sqrt(20)`, оскільки функція `sqrt()`, яка повертає квадратний корінь числа, є статичною.

Модифікатори доступу. Інкапсуляція

Модифікатори доступу дають змогу задати допустиму сферу видимості для членів класу, тобто контекст, у якому можна вживати цю змінну або метод. У Java використовуються такі модифікатори доступу:

- **public**: публічний, загальнодоступний клас або член класу. Поля і методи, оголошені з модифікатором **public**, видно іншим класам із поточного пакета і з зовнішніх пакетів;
- **private**: закритий клас або член класу, протилежність модифікатору **public**. Закритий клас або член класу доступний тільки з коду в тому ж класі;
- **protected**: такий клас або член класу доступний з будь-якого місця в поточному класі або пакеті або в похідних класах, навіть якщо вони знаходяться в інших пакетах;
- Модифікатор за замовчуванням (**default**). Відсутність модифікатора у поля або методу класу передбачає застосування до нього модифікатора за замовчуванням. Такі поля або методи видно всім класам у поточному пакеті.

РОЗГЛЯНЕМО ПРИКЛАД

Модифікатори доступу. Інкапсуляція

```
public class Example {
    public static void main(String[] args) {
        Person kate = new Person( name: "Andrew", age: 30, address: "Baker Street", phone: "+12334567");
        kate.displayName(); // fine, public
        kate.displayAge(); // fine, default
        kate.displayPhone(); // fine, protected
        //kate.displayAddress(); // ! Error, private

        System.out.println(kate.name); // fine, default
        System.out.println(kate.address); //fine, public
        System.out.println(kate.age); // fine, protected
        //System.out.println(kate.phone); // ! Error, private
    }
}

2 usages
class Person {
    3 usages
    String name;
    3 usages
    protected int age;
    3 usages
    public String address;
    2 usages
    private String phone;

    1 usage
    public Person(String name, int age, String address, String phone) {
        this.name = name;
        this.age = age;
        this.address = address;
        this.phone = phone;
    }

    1 usage
    public void displayName() {System.out.printf("Name: %s \n", name);}
    1 usage
    void displayAge() {System.out.printf("Age: %d \n", age);}
    no usages
    private void displayAddress() {System.out.printf("Address: %s \n", address);}
    1 usage
    protected void displayPhone() {System.out.printf("Phone: %s \n", phone);}
}
```

У цьому випадку обидва класи розташовані в одному пакеті - пакеті за замовчуванням, тому в класі Program ми можемо використовувати всі методи і змінні класу Person, які мають модифікатор за замовчуванням, public і protected. А поля і методи з модифікатором private у класі Program не будуть доступні.

Якби клас Program розташовувався б в іншому пакеті, то йому були б доступні тільки поля і методи з модифікатором public.

Модифікатор доступу має передувати решті частини визначення змінної або методу.

Модифікатори доступу. Інкапсуляція

Термін «**інкапсуляція**», він же перший принцип ООП, має два трактування. Найчастіше фахівці використовують цей термін тільки в одному значенні, забуваючи про інше, а це невірно.

Перше трактування – в один об'єкт або клас об'єднуються і дані, і методи, які працюють з цими даними.

Друге трактування – інкапсуляція це приховування внутрішньої структури об'єкта від зовнішніх впливів. Всі зміни стану об'єкта відбуваються тільки за допомогою звернень до методів самого об'єкту.

Особисто я розумію під терміном «інкапсуляція» обидві ці трактування – і з'єднання методів з даними, і приховування даних від зовнішніх впливів, зміна їх тільки через звернення до методів самого об'єкту. Справа в тому, що друга трактування без першої не працює зовсім. Що ви будете приховувати, якщо у вас в об'єкті не буде методів і даних?

До чого відноситься принцип інкапсуляції?

З точки зору ООП, об'єктом є не тільки клас або об'єкт, або як це називається в вашій мові програмування, а й інші структури, більш високого рівня, а саме: packages, namespaces, модулі, сабмодулі (submodules), підсистеми і вся система цілком. Якщо розглядати інкапсуляцію як приховування, ми розуміємо, що наш об'єкт не повинен змінюватися нічим зовні, крім його ж методів.

Модифікатори доступу. Інкапсуляція

Так, як правило, коли ми згадуємо про інкапсуляцію, замість безпосереднього застосування полів ми використовуємо методи доступу.

І потім замість безпосередньої роботи з полями `name` і `age` у класі `Person` ми будемо працювати з методами, які встановлюють і повертають значення цих полів. Методи `setName`, `setAge` і подібні ще називають м'ютейтерами (*mutator*), оскільки вони змінюють значення поля. А методи `getName`, `getAge` і подібні називають аксесерами (*accessor*), оскільки з їхньою допомогою ми отримуємо значення поля. Причому в ці методи ми можемо вкласти додаткову логіку.

```
class Person{  
    3 usages  
    private String name;  
    3 usages  
    private int age = 1;  
    1 usage  
    public Person(String name, int age){  
        setName(name);  
        setAge(age);  
    }  
    public String getName(){return this.name;}  
    public void setName(String name){this.name = name;}  
    no usages  
    public int getAge(){return this.age;}  
    1 usage  
    public void setAge(int age){  
        if(age > 0 && age < 110)  
            this.age = age;  
    }  
}
```

Статичні члени та модифікатор static

Крім звичайних методів і полів клас може мати статичні поля, методи, константи та ініціалізатори. Наприклад, головний клас програми має метод `main`, який є статичним.

Статичні поля

Під час створення об'єктів класу для кожного об'єкта створюється своя копія нестатичних звичайних полів. А статичні поля є загальними для всього класу. Тому вони можуть використовуватися без створення об'єктів класу. Наприклад, створимо статичну змінну:

```
class Person{  
    2 usages  
    private int id;  
    1 usage  
    static int counter=1;  
  
    1 usage  
    Person(){  
        id = counter++;  
    }  
    no usages  
    public void displayId(){  
        System.out.printf("Id: %d \n", id);  
    }  
}
```

Статичні члени та модифікатор `static`

Клас **Person** містить статичну змінну ***counter***, яка збільшується в конструкторі та її значення присвоюється змінній ***id***. Тобто при створенні кожного нового об'єкта **Person** ця змінна буде збільшуватися, тому в кожного нового об'єкта **Person** значення поля ***id*** буде на 1 більше ніж у попереднього. Оскільки змінна ***counter*** статична, то ми можемо звернутися до неї в програмі за ім'ям класу:

```
System.out.println(Person.counter); // отримуємо значення
Person.counter = 8; // змінюємо значення
```

Статичні константи

Також статичними бувають константи, які є загальними для всього класу.

```
class Math{
    public static final double PI = 3.14;
}
```

Статичні ініціалізатори

Статичні ініціалізатори призначені для ініціалізації статичних змінних, або для виконання таких дій, які виконуються під час створення найпершого об'єкта. Наприклад, визначимо статичний ініціалізатор:

Статичні члени та модифікатор static

Статичний ініціалізатор визначається як звичайний, тільки перед ним ставиться ключове слово `static`. У цьому випадку в статичному ініціалізаторі ми встановлюємо початкове значення статичного поля `counter` і виводимо на консоль повідомлення.

Варто враховувати, що виклик статичного ініціалізатора здійснюється після завантаження класу і фактично до створення найпершого об'єкта класу.

```
class Person{  
    2 usages  
    private int id;  
    2 usages  
    static int counter;  
  
    static{  
        counter = 105;  
        System.out.println("Static initializer");  
    }  
    1 usage  
    Person(){  
        id=counter++;  
        System.out.println("Constructor");  
    }  
    no usages  
    public void displayId(){  
        System.out.printf("Id: %d \n", id);  
    }  
}
```

Статичні члени та модифікатор static

Статичні методи

Статичні методи також належать до всього класу загалом. Наприклад, у прикладі вище статична змінна `counter` була доступна ззовні, і ми могли змінити її значення поза класом `Person`. Зробимо її недоступною для зміни ззовні, але доступною для читання. Тепер статична змінна недоступна ззовні, вона приватна. А її значення виводиться за допомогою статичного методу `displayCounter`. Для звернення до статичного методу використовується ім'я класу: `Person.displayCounter()`.

Під час використання статичних методів треба враховувати обмеження: у статичних методах ми можемо викликати тільки інші статичні методи і використовувати тільки статичні змінні.

```
class Person{
    2 usages
    private int id;
    2 usages
    private static int counter = 1;

    1 usage
    Person(){
        id = counter++;
    }

    no usages
    public static void displayCounter(){
        System.out.printf("Counter: %d \n", counter);
    }

    no usages
    public void displayId(){
        System.out.printf("Id: %d \n", id);
    }
}
```

Об'єкти як параметри методів

Об'єкти класів, як і дані примітивних типів, можуть передаватися в методи. Однак у цьому випадку є одна особливість - під час передачі об'єктів як значення передається копія посилання на ділянку в пам'яті, де розташований цей об'єкт. Розглянемо невеликий приклад. Нехай у нас є такий клас **Person**:

```
public class MyExample{
    public static void main(String[] args) {
        Person kate = new Person( name: "Kate");
        System.out.println(kate.getName());    // Kate
        changeName(kate);
        System.out.println(kate.getName());    // Alice
    }
    1 usage
    static void changeName(Person p){
        p.setName("Alice");
    }
}
3 usages
class Person{
    3 usages
    private String name;
    1 usage
    Person(String name){
        this.name = name;
    }
    public void setName(String name){
        this.name = name;
    }
    public String getName(){
        return this.name;
    }
}
```

Тут у метод **changeName** передається об'єкт **Person**, у якого змінюється ім'я. Оскільки в метод буде передаватися копія посилання на ділянку пам'яті, в якій знаходиться об'єкт **Person**, то змінна *kate* і параметр *p* методу **changeName** вказуватимуть на один і той самий об'єкт у пам'яті. Тому після виконання методу в об'єкта *kate*, який передається в метод, буде змінено ім'я з "**Kate**" на "**Alice**".

Внутрішні та вкладені класи

Класи можуть бути **вкладеними (nested)**, тобто можуть бути визначені всередині інших класів. Одним випадком вкладених класів є **внутрішні класи (inner class)**. Наприклад, є клас Person, усередині якого визначено клас Account.

Внутрішній клас поводить себе як звичайний клас за тим винятком, що його об'єкти можуть бути створені тільки всередині зовнішнього класу.

```
public class MyExample{
    public static void main(String[] args) {
        Person tom = new Person( name: "Tom", password: "qwerty");
        tom.displayPerson();
        tom.account.displayAccount();
    }
}
3 usages
class Person{
    3 usages
    private String name;
    3 usages
    Account account;
    1 usage
    Person(String name, String password){
        this.name = name;
        account = new Account(password);
    }
    1 usage
    public void displayPerson(){
        System.out.printf("Person \t Name: %s \t Password: %s \n", name, account.password);
    }
    2 usages
    public class Account{
        3 usages
        private String password;
        1 usage
        Account(String pass){
            this.password = pass;
        }
        1 usage
        void displayAccount(){
            System.out.printf("Account Login: %s \t Password: %s \n", Person.this.name, password);
        }
    }
}
```

Внутрішні та вкладені класи

Внутрішній клас має доступ до всіх полів зовнішнього класу, у тому числі закритих за допомогою модифікатора **private**. Аналогічно зовнішній клас має доступ до всіх членів внутрішнього класу, в тому числі до полів і методів з модифікатором **private**.

Посилання на об'єкт зовнішнього класу з внутрішнього класу можна отримати за допомогою виразу **Зовнішній_клас.this**, наприклад, **Person.this**.

Об'єкти внутрішніх класів можуть бути створені тільки в тому класі, в якому внутрішні класи визначені. В інших зовнішніх класах об'єкти внутрішнього класу створити не можна.

Ще однією особливістю внутрішніх класів є те, що їх можна оголосити всередині будь-якого контексту, зокрема всередині методу і навіть у циклі:

```
public class MyExample{
    public static void main(String[] args) {
        Person tom = new Person( "Tom");
        tom.setAccount("qwerty");
    }
}
2 usages

class Person{
    2 usages
    private String name;
    1 usage
    Person(String name){this.name = name;}
    1 usage
    public void setAccount (String password){
        2 usages
        class Account{
            1 usage
            void display(){
                System.out.printf("Account Login: " +
                    "%s \t Password: %s \n", name, password);
            }
        }
        Account account = new Account();
        account.display();
    }
}
```


Внутрішні та вкладені класи

Статичні вкладені класи

Крім внутрішніх класів також можуть бути статичні вкладені класи. Статичні вкладені класи дають змогу приховати деяку комплексну інформацію всередині зовнішнього класу.

Тут визначено вкладений клас для зберігання даних про обчислення факторіалу. Основні дії виконує метод `getFactorial`, який повертає об'єкт вкладеного класу.

```
class Math{
    2 usages
    public static class Factorial{
        2 usages
        private int result;
        2 usages
        private int key;
        1 usage
        public Factorial(int number, int x){
            result=number;
            key = x;
        }
        no usages
        public int getResult(){
            return result;
        }
        no usages
        public int getKey(){
            return key;
        }
    }
    no usages
    public static Factorial getFactorial(int x){
        int result=1;
        for (int i = 1; i <= x; i++){
            result *= i;
        }
        return new Factorial(result, x);
    }
}
```

Успадкування

Одним із ключових аспектів об'єктно-орієнтованого програмування є **успадкування**.

Успадкування в Java можна визначити як техніку або процес, при якому один об'єкт класу набуває поведінки та властивостей іншого об'єкта.

Це робиться шляхом успадкування класу або встановлення зв'язку між двома класами. За допомогою успадкування можна розширити функціонал уже наявних класів за рахунок додавання нового функціоналу або зміни старого. Наприклад, є такий клас Person, що описує окрему людину:

```
class Person {  
    3 usages  
    String name;  
    public String getName(){ return name; }  
  
    no usages  
    public Person(String name){  
        this.name=name;  
    }  
  
    no usages  
    public void display(){  
        System.out.println("Name: " + name);  
    }  
}
```

```
no usages  
class Employee extends Person {  
    no usages  
    public Employee(String name) {  
        super(name);  
    }  
}
```

І, можливо, згодом ми захочемо додати ще один клас, який описує співробітника підприємства - клас Employee. Оскільки цей клас реалізує той самий функціонал, що й клас Person, оскільки працівник - це також і людина, то було б раціонально зробити клас Employee похідним (спадкоємцем, підкласом) від класу Person, який, своєю чергою, називається базовим класом, батьком або суперкласом.

Успадкування

Щоб оголосити один клас спадкоємцем від іншого, треба використовувати після імені класу-спадкоємця ключове слово **extends**, після якого йде ім'я базового класу. Для класу **Employee** базовим є **Person**, і тому клас **Employee** успадковує всі ті самі поля та методи, які є в класі **Person**.

Якщо в базовому класі визначено конструктори, то в конструкторі похідного класу необхідно викликати один із конструкторів базового класу за допомогою ключового слова **super**. Наприклад, клас **Person** має конструктор, який приймає один параметр. Тому в класі **Employee** у конструкторі потрібно викликати конструктор класу **Person**. Тобто виклик **super(name)** представлятиме виклик конструктора класу **Person**.

Під час виклику конструктора після слова **super** у дужках іде перерахування переданих аргументів. При цьому виклик конструктора базового класу має йти на самому початку в конструкторі похідного класу. Таким чином, встановлення імені співробітника делегується конструктору базового класу.

Причому навіть якщо похідний клас ніякої іншої роботи не виконує в конструкторі, як у прикладі вище, все одно необхідно викликати конструктор базового класу.

Успадкування

```
public class MyExample{
    public static void main(String[] args) {
        Person tom = new Person( name: "Tom");
        tom.display();
        Employee sam = new Employee( name: "Sam");
        sam.display();
    }
}

3 usages 1 inheritor
class Person {
    3 usages
    String name;
    public String getName(){ return name; }
    2 usages
    public Person(String name){
        this.name=name;
    }
    2 usages
    public void display(){
        System.out.println("Name: " + name);
    }
}

2 usages
class Employee extends Person{
    1 usage
    public Employee(String name){
        super(name);
    }
}
```

Похідний клас має доступ до всіх методів і полів базового класу (навіть якщо базовий клас перебуває в іншому пакеті), крім тих, що визначені з модифікатором **private**. При цьому похідний клас також може додавати свої поля та методи.

```
public class MyExample{
    public static void main(String[] args) {
        Employee sam = new Employee( name: "Sam", company: "Microsoft");
        sam.display(); // Sam
        sam.work();    // Sam works in Microsoft
    }
}

1 usage 1 inheritor
class Person {
    3 usages
    String name;
    public String getName(){ return name; }
    1 usage
    public Person(String name){
        this.name=name;
    }
    1 usage
    public void display(){
        System.out.println("Name: " + name);
    }
}

2 usages
class Employee extends Person{
    2 usages
    String company;
    1 usage
    public Employee(String name, String company) {
        super(name);
        this.company=company;
    }
    1 usage
    public void work(){
        System.out.printf("%s works in %s \n", getName(), company);
    }
}
```

У цьому випадку клас Employee додає поле company, яке зберігає місце роботи співробітника, а також метод work.

Успадкування

Перевизначення методів

Похідний клас може визначати свої методи, а може перевизначати методи, які успадковані від базового класу. Наприклад, перевизначимо в класі Employee метод display.

Перед перевизначуванням методом вказується анотація **@Override**. Ця анотація в принципі необов'язкова.

При перевизначенні методу він повинен мати рівень доступу не менший, ніж рівень доступу в базовому класу. Наприклад, якщо в базовому класі метод має модифікатор public, то і в похідному класі метод повинен мати модифікатор public.

```
public class MyExample{
    public static void main(String[] args) {
        Employee sam = new Employee( name: "Sam", company: "Microsoft");
        sam.display(); // Sam
        // Works in Microsoft
    }
}

1 usage 1 inheritor
class Person {
    3 usages
    String name;
    public String getName(){ return name; }
    1 usage
    public Person(String name){
        this.name=name;
    }
    1 usage 1 override
    public void display(){
        System.out.println("Name: " + name);
    }
}

2 usages
class Employee extends Person{
    2 usages
    String company;
    1 usage
    public Employee(String name, String company) {
        super(name);
        this.company=company;
    }
    1 usage
    @Override
    public void display(){
        System.out.printf("Name: %s \n", getName());
        System.out.printf("Works in %s \n", company);
    }
}
```

Успадкування

Однак у минулому випадку ми бачимо, що частина методу `display` у `Employee` повторює дії з методу `display` базового класу. Тому ми можемо скоротити клас `Employee`.

```
class Employee extends Person{  
    2 usages  
    String company;  
    1 usage  
    public Employee(String name, String company) {  
        super(name);  
        this.company=company;  
    }  
    2 usages  
    @Override  
    public void display(){  
        super.display();  
        System.out.printf("Works in %s \n", company);  
    }  
}
```

За допомогою ключового слова `super` ми також можемо звернутися до реалізації методів базового класу.

Успадкування

Заборона успадкування

Хоча успадкування дуже цікавий і ефективний механізм, але в деяких ситуаціях його застосування може бути небажаним. І в цьому випадку можна заборонити успадкування за допомогою ключового слова `final`. Наприклад:

```
public final class Person {}
```

Якби клас `Person` був би визначений таким чином, то наступний код був би помилковим і не спрацював, оскільки ми тим самим заборонили успадкування:

```
class Employee extends Person{}
```

Крім заборони успадкування можна також заборонити перевизначення окремих методів:

```
public final void display(){  
    System.out.println("Ім'я: " + name);  
}
```

Успадкування

Динамічна диспетчеризація методів

Успадкування і можливість перевизначення методів відкривають нам великі можливості. Насамперед ми можемо передати змінній суперкласу посилання на об'єкт підкласу:

```
Person sam = new Employee("Sam", "Oracle");
```

Оскільки Employee успадковується від Person, то об'єкт Employee є водночас і об'єктом Person. Грубо кажучи, будь-який працівник підприємства одночасно є людиною.

Однак незважаючи на те, що змінна представляє об'єкт Person, віртуальна машина бачить, що в реальності вона вказує на об'єкт Employee. Тому під час виклику методів у цього об'єкта буде викликатися та версія методу, яка визначена в класі Employee, а не в Person. Наприклад:

Успадкування

```
public class MyExample{
    public static void main(String[] args) {
        Person tom = new Person( name: "Tom");
        tom.display();
        Person sam = new Employee( name: "Sam", company: "Oracle");
        sam.display();
    }
}
4 usages 1 inheritor
class Person {
    3 usages
    String name;
    public String getName() { return name; }
    2 usages
    public Person(String name){this.name=name;}

    2 usages 1 override
    public void display(){
        System.out.printf("Person %s \n", name);
    }
}
1 usage
class Employee extends Person{
    2 usages
    String company;
    1 usage
    public Employee(String name, String company) {
        super(name);
        this.company = company;
    }
    2 usages
    @Override
    public void display(){
        System.out.printf("Employee %s works in %s \n", super.getName(), company);
    }
}
```

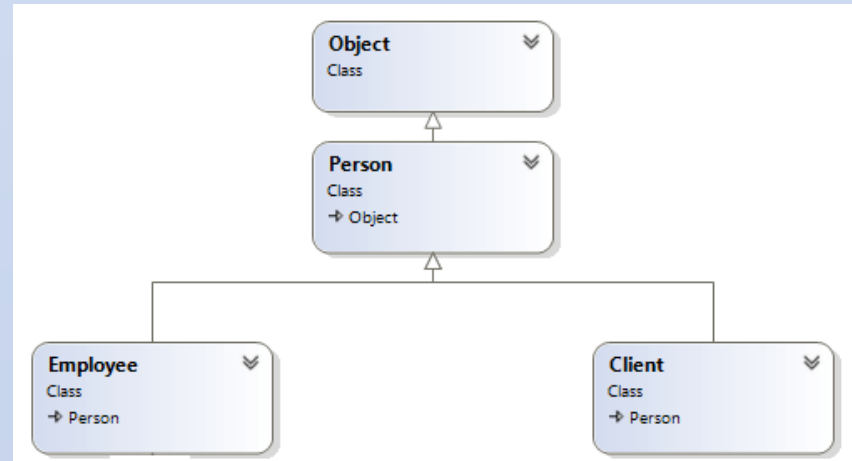
Під час виклику перевизначеного методу віртуальна машина динамічно знаходить і викликає саме ту версію методу, яка визначена в підкласі. Цей процес ще називається dynamic method lookup або динамічний пошук методу чи динамічна диспетчеризація методів.

Ієрархія успадкування та перетворення типів

Раніше ми говорили про перетворення об'єктів простих типів. Однак з об'єктами класів усе відбувається трохи по-іншому. Припустимо, у нас є така ієрархія класів:

```
public class MyExample{  
    public static void main(String[] args) {  
    }  
}  
  
2 usages  2 inheritors  
class Person {  
  
}  
  
no usages  
class Employee extends Person{  
  
}  
  
no usages  
class Client extends Person{  
  
}
```

У цій ієрархії класів можна простежити такий ланцюг успадкування: Object (усі класи неявно успадковуються від типу Object) -> Person -> Employee|Client.



Ієрархія успадкування та перетворення типів

Суперкласи зазвичай розміщуються вище за підкласи, тому на вершині успадкування знаходиться клас **Object**, а в самому низу **Employee** і **Client**. Об'єкт підкласу також представляє об'єкт суперкласу. Тому в програмі ми можемо написати таким чином:

```
Object tom = new Person( name: "Tom");
Object sam = new Employee( name: "Sam", company: "Oracle");
Object kate = new Client( name: "Kate", bank: "DeutscheBank", sum: 2000);
Person bob = new Client( name: "Bob", bank: "DeutscheBank", sum: 3000);
Person alice = new Employee( name: "Alice", company: "Google");
```

Це так зване висхідне перетворення (від підкласу вниз до суперкласу вгорі ієрархії) або **upcasting**. Таке перетворення здійснюється автоматично. Зворотне не завжди вірне. Наприклад, об'єкт **Person** не завжди є об'єктом **Employee** або **Client**. Тому низхідне перетворення або **downcasting** від суперкласу до підкласу автоматично не виконується. У цьому випадку нам треба використовувати операцію перетворення типів.

Ієрархія успадкування та перетворення типів

У цьому випадку змінна `sam` приводиться до типу `Employee`. І потім через об'єкт `emp` ми можемо звернутися до функціоналу об'єкта `Employee`. Ми можемо перетворити об'єкт `Employee` по всій прямій лінії успадкування від `Object` до `Employee`.

Приклади низхідних перетворень:

```
public class MyExample {  
    public static void main(String[] args) {  
        Object sam = new Employee( name: "Sam", company: "Oracle");  
        // низхідне перетворення від Object до типу Employee  
        Employee emp = (Employee)sam;  
        emp.display();  
        System.out.println(emp.getCompany());  
    }  
}
```

```
public static void main(String[] args) {  
    Object kate = new Client( name: "Kate", bank: "DeutscheBank", sum: 2000);  
    ((Person)kate).display();  
  
    Object sam = new Employee( name: "Sam", company: "Oracle");  
    ((Employee)sam).display();  
}
```

Ієрархія успадкування та перетворення типів

Оператор instanceof

У прикладі раніше ми явно бачимо, що змінна `kate` - це посилання на об'єкт `Client`, а не `Employee`. Однак нерідко дані приходять ззовні, і ми можемо точно не знати, який саме об'єкт ці дані представляють. Відповідно виникає велика ймовірність зіткнутися з помилкою. І перед тим, як провести перетворення типів, ми можемо перевірити, а чи можемо ми виконати приведення за допомогою оператора `instanceof`:

```
public static void main(String[] args) {  
    Object kate = new Client( name: "Kate", bank: "DeutscheBank", sum: 2000);  
    if(kate instanceof Employee){  
        Employee employeeKate = (Employee) kate;  
        employeeKate.display();  
    }  
    else{  
        System.out.println("Conversion is invalid");  
    }  
}
```

Ієрархія успадкування та перетворення типів

Вираз `kate instanceof Employee` перевіряє, чи є змінна `kate` об'єктом типу `Employee`. Але оскільки в цьому випадку явно не є, то така перевірка поверне значення `false`, і перетворення не спрацює. А вираз `kate instanceof Client` повернув би `true`:

```
public static void main(String[] args) {
    Object kate = new Client( name: "Kate", bank: "DeutscheBank", sum: 2000);
    if(kate instanceof Client){
        Client clientKate = (Client) kate;
        clientKate.display();
    }
    else{
        System.out.println("Conversion is invalid");
    }
}
```

```
Object kate = new Client( name: "Kate", bank: "DeutscheBank", sum: 2000)
if(kate instanceof Client clientKate){
    clientKate.display();
}
else{
    System.out.println("Conversion is invalid");
}
```

Слід зазначити, що починаючи з версії Java 16 ми можемо спростити перетворення типів таким чином: Вираз `kate instanceof Client clientKate` перевіряє, чи представляє змінна `kate` клас `Client`, і якщо представляє (тобто оператор `instanceof` повертає `true`), то створює змінну `clientKate` типу `Client`. І надалі ми можемо використовувати цю змінну `clientKate` і робити з нею різні операції.

Клас Object і його методи

Хоча ми можемо створити звичайний клас, який не є спадкоємцем, але фактично всі класи успадковуються від класу **Object**. Усі інші класи, навіть ті, які ми додаємо до свого проєкту, є неявно похідними від класу **Object**. Тому всі типи і класи можуть реалізувати ті методи, які визначені в класі **Object**. Розглянемо ці методи.

Метод toString

Метод toString служить для отримання подання даного об'єкта у вигляді рядка. Під час спроби вивести строкове представлення якого-небудь об'єкта, як правило, буде виводитися повне ім'я класу (наприклад **Person@7958403b**). Тому потрібно перевизначати метод, наприклад:

```
@Override
public String toString(){
    return "Person " + name;
}
```

Клас Object і його методи

Метод *hashCode*

Метод `hashCode` дає змогу задати деяке числове значення, яке відповідатиме цьому об'єкту або його хеш-код. За цим числом, наприклад, можна порівнювати об'єкти. Наприклад, виведемо подання об'єкта класу `Person`:

```
public class MyExample{  
    public static void main(String[] args) {  
        Person andrii = new Person( name: "Andrii");  
        System.out.println(andrii.hashCode()); // 2036368507  
    }  
}
```

Але ми можемо задати свій алгоритм визначення хеш-коду об'єкта:

```
@Override  
public int hashCode(){  
    return 10 * name.hashCode() + 20456;  
}
```


Клас Object і його методи

Метод *getClass*

Метод `getClass` дає змогу отримати тип цього об'єкта:

```
public static void main(String[] args) {  
    Person andrii = new Person( name: "Andrii");  
    System.out.println(andrii.getClass()); // class Person  
}
```

Метод *equals*

Метод `equals` порівнює два об'єкти на рівність:

```
@Override  
public boolean equals(Object obj){  
    if (!(obj instanceof Person)) return false;  
    Person p = (Person)obj;  
    return this.name.equals(p.name);  
}
```

Клас Object і його методи

Метод **equals** приймає як параметр об'єкт будь-якого типу, який ми потім приводимо до поточного, якщо вони є об'єктами одного класу.

Оператор **instanceof** дає змогу з'ясувати, чи є переданий як параметр об'єкт об'єктом певного класу, у цьому випадку класу Person. Якщо об'єкти належать до різних класів, то їхнє порівняння не має сенсу, і повертається значення false.

Потім порівнюємо за іменами. Якщо вони збігаються, повертаємо true, що буде говорити, що об'єкти рівні.

Поліморфізм

Поліморфізм вважається однією з важливих особливостей об'єктно-орієнтованого програмування. Поліморфізм дозволяє нам виконувати одну дію різними способами. Іншими словами, поліморфізм дозволяє визначити один інтерфейс і мати безліч його реалізацій. Слово "полі" означає багато, а "морфи" - форми, тобто багато форм.

В Java поліморфізм в основному поділяється на два типи:

- Поліморфізм під час компіляції;
- Поліморфізм під час виконання.

Поліморфізм часу компіляції - також відомий як статичний поліморфізм. Цей тип поліморфізму досягається перевантаженням функцій або операторів.

Зауваження: Java не підтримує перевантаження операторів, як наприклад C++

Поліморфізм

Перевантаження методів (вже розглядали на минулих лекціях)

Коли існує декілька функцій з однаковими іменами, але різними параметрами, то кажуть, що ці функції перевантажені. Функції можуть бути перевантажені зміною кількості аргументів та/або зміною типу аргументів.

Поліморфізм виконання

Він також відомий як **динамічна диспетчеризація методів**. Це процес, в якому виклик функції до перевизначеного методу вирішується під час виконання. Цей тип поліморфізму досягається за допомогою перевизначення методів. З іншого боку, перевизначення методу відбувається, коли похідний клас має визначення однієї з функцій-членів базового класу. Кажуть, що ця базова функція перевизначена.

Поліморфізм

У цій програмі, коли створюється об'єкт дочірнього класу, викликається метод всередині дочірнього класу. Це відбувається тому, що метод батьківського класу перевизначається дочірнім класом. Оскільки метод перевизначається, то цей метод має більший пріоритет, ніж батьківський метод всередині дочірнього класу. Отже, виконується тіло всередині дочірнього класу.

```
class Parent {  
    2 usages  2 overrides  
    void print() { System.out.println("Parent class");}  
}  
  
1 usage  
class FirstChild extends Parent {  
    2 usages  
    void print() { System.out.println("FirstChild"); }  
}  
  
1 usage  
class SecondChild extends Parent {  
    2 usages  
    void print() {System.out.println("SecondChild");}  
}  
  
class Example {  
  
    public static void main(String[] args)  
    {  
        Parent a;  
  
        a = new FirstChild();  
        a.print(); // FirstChild  
  
        a = new SecondChild();  
        a.print(); // SecondChild  
    }  
}
```

Додаткова інформація

Застосування ООП дає змогу створювати більш гнучкі, модульні та масштабовані програми. Класи та об'єкти можуть використовуватися для моделювання різних реальних об'єктів і процесів, що полегшує проектування та розробку програмного забезпечення.

Композиція та **агрегація** є двома поняттями об'єктно-орієнтованого програмування, які описують відносини між об'єктами. Обидва ці поняття стосуються використання об'єктів для створення більших структур або компонентів у програмі.

Агрегація - це відношення між об'єктами, за якого один об'єкт є власником іншого об'єкта і може містити його в собі. У цьому випадку об'єкт, який містить інший об'єкт, називається агрегатором, а об'єкт, який міститься всередині іншого об'єкта, називається агрегатом. Агрегація дає змогу створювати більші структури, об'єднуючи безліч об'єктів в один об'єкт.

Додаткова інформація

Прикладом агрегації може слугувати об'єкт "Команда", який містить кілька об'єктів "Гравець". Команда може додавати і видаляти гравців залежно від потреб команди. Однак, якщо об'єкт "Команда" буде видалено, то об'єкти "Гравець" усе ще існуватимуть.

Композиція - це відношення між об'єктами, за якого один об'єкт є частиною іншого об'єкта і не може існувати незалежно від нього. У цьому випадку об'єкт, який містить інший об'єкт, називається контейнером, а об'єкт, який міститься всередині іншого об'єкта, називається компонентом. Композиція дозволяє створювати більші об'єкти, використовуючи безліч менших об'єктів.

Прикладом композиції може слугувати об'єкт "Двигун", який є частиною об'єкта "Автомобіль". Якщо об'єкт "Автомобіль" буде видалено, то об'єкт "Двигун" також буде видалено.

В обох випадках об'єкти можуть мати інші відносини, як-от успадкування, і можуть містити методи та властивості для їхньої обробки. Однак, у випадку композиції, якщо батьківський об'єкт видаляється, то всі вкладені об'єкти також будуть видалені, тоді як у випадку агрегації цього не відбувається. Важливо розуміти відмінності між композицією та агрегацією, щоб правильно використовувати їх у своїх програмах і створювати більш ефективні та гнучкі об'єктні моделі.

Взагалі, щоб краще розуміти навіщо вам узагалі розуміти ООП, то потрібно звернутися до **патернів проектування**.

Контрольні питання

1. Дайте визначення поняттю «ООП».
2. Дайте визначення поняттю «інкапсуляція».
3. Дайте визначення поняттю «поліморфізм».
4. Дайте визначення поняттю «успадкування».
5. Для чого потрібні конструктори в класі.
6. Які існують модифікатори доступу в Java.
7. Які є недоліки в успадкуванні.
8. Визначте типи поліморфізму в Java. Охарактеризуйте їх.

The END
Lec 5.

.NET та JAVA #6

Абстрактні класи та інтерфейси

1. Абстрактні класи
2. Інтерфейси
3. Інтерфейси в механізмі зворотного виклику
4. Перерахування enum
5. Узагальнення (Generics)
6. Обмеження узагальнень

Абстрактні класи

Крім звичайних класів у Java є абстрактні класи. Абстрактний клас схожий на звичайний клас. В абстрактному класі також можна визначити поля і методи, але водночас не можна створити об'єкт або екземпляр абстрактного класу. Абстрактні класи покликані надавати базовий функціонал для класів-спадкоємців. А похідні класи вже реалізують цей функціонал.

При визначенні абстрактних класів використовується ключове слово **abstract**:

```
public abstract class Human{  
  
    private String name;  
  
    public String getName() { return name; }  
}
```

Difference between Abstract Classes & Interfaces			
Abstract Classes		Interfaces	
Abstract classes are fast,	Speed	Interfaces are slow.	
Abstract classes can extend only one class.	Multiple Inheritance	Interface can implement several interfaces.	
You can define fields as well as constants.	Defined Fields	You cannot define fields in an interface.	
A single abstract class can extend one & only one interface.	Extension Limit	A single interface can extend multiple interfaces.	

Абстрактні класи

Але головна відмінність полягає в тому, що ми не можемо використовувати конструктор абстрактного класу для створення його об'єкта. Наприклад, у такий спосіб:

```
Human andrii = new Human();
```

Крім звичайних методів абстрактний клас може містити абстрактні методи. Такі методи визначаються за допомогою ключового слова `abstract` і не мають жодної реалізації:

```
public abstract void display();
```

Похідний клас зобов'язаний перевизначити та реалізувати всі абстрактні методи, які є в базовому абстрактному класі. Також слід враховувати, що якщо клас має хоча б один абстрактний метод, то цей клас має бути визначений як абстрактний.

Абстрактні класи

Навіщо потрібні абстрактні класи?

Припустімо, ми робимо програму для обслуговування банківських операцій і визначаємо в ній три класи:

- **Person**, що описує людину;
- **Employee**, що описує банківського службовця;
- **Client**, що представляє клієнта банку.

Очевидно, що класи **Employee** і **Client** будуть похідними від класу **Person**, тому що обидва класи мають деякі загальні поля і методи. І оскільки всі об'єкти будуть представляти або співробітника, або клієнта банку, то безпосередньо ми від класу **Person** створювати об'єкти не будемо. Тому має сенс зробити його абстрактним.

Абстрактні класи

```
public class Program{
    public static void main(String[] args) {
        Employee sam = new Employee( name: "Taras", company: "Luchan Brothers");
        sam.display();
        Client bob = new Client( name: "Stepan", company: "Luchan Brothers");
        bob.display();
    }
}

2 usages 2 inheritors
abstract class Person {
    2 usages
    private String name;
    public String getName() { return name; }
    2 usages
    public Person(String name){ this.name=name;}
    2 usages 2 implementations
    public abstract void display();
}

2 usages
class Employee extends Person{
    2 usages
    private String bank;
    1 usage
    public Employee(String name, String company) {
        super(name);
        this.bank = company;
    }
    2 usages
    public void display(){ System.out.printf("Employee Name: %s \t Bank: %s \n", super.getName(), bank);}
}

2 usages
class Client extends Person {
    2 usages
    private String bank;
    1 usage
    public Client(String name, String company) {
        super(name);
        this.bank = company;
    }
    2 usages
    public void display(){System.out.printf("Client Name: %s \t Bank: %s \n", super.getName(), bank);}
}
```

Абстрактні класи

Іншим хрестоматійним прикладом є система геометричних фігур. У реальності не існує геометричної фігури як такої. Є коло, прямокутник, квадрат, але просто фігури немає. Проте ж і коло, і прямокутник мають щось спільне і є фігурами:

```
// абстрактний клас фігури
1 usage 2 inheritors
abstract class Figure{
    1 usage
    float x; // x-координата точки
    1 usage
    float y; // y-координата точки
    1 usage
    Figure(float x, float y){
        this.x=x;
        this.y=y;
    }
    // абстрактний метод для отримання периметра
    no usages
    public abstract float getPerimeter();
    // абстрактний метод для отримання площі
    no usages 2 implementations
    public abstract float getArea();
}
// похідний клас прямокутника
no usages
class Rectangle extends Figure {
    3 usages
    private float width;
    3 usages
    private float height;
    // конструктор зі зверненням до конструктора класу Figure
    no usages
    Rectangle(float x, float y, float width, float height){
        super(x,y);
        this.width = width;
        this.height = height;
    }
    no usages
    public float getPerimeter(){ return width * 2 + height * 2;}
    no usages
    public float getArea(){return width * height;}
}
```

Інтерфейси

Механізм успадкування дуже зручний, але він має свої обмеження. Зокрема ми можемо успадковувати тільки від одного класу, на відміну, наприклад, від мови C++, де є множинне успадкування.

У мові Java подібну проблему частково дають змогу вирішити **інтерфейси**. **Інтерфейси** визначають деякий функціонал, який не має конкретної реалізації, який потім реалізують класи, що застосовують ці інтерфейси. І один клас може застосувати безліч інтерфейсів.

Щоб визначити інтерфейс, використовується ключове слово `interface`. Наприклад:

```
interface Printable{  
    void print();  
}
```


Інтерфейси

Цей інтерфейс називається **Printable**. Інтерфейс може визначати константи та методи, які можуть мати, а можуть і не мати реалізації. Методи без реалізації схожі на абстрактні методи абстрактних класів. Так, у цьому випадку оголошено один метод, який не має реалізації.

Усі методи інтерфейсу **не мають модифікаторів доступу**, але фактично за замовчуванням доступ **public**, оскільки мета інтерфейсу - визначення функціоналу для реалізації його класом. Тому весь функціонал має бути відкритий для реалізації.

Щоб клас застосував інтерфейс, треба використовувати ключове слово `implements`, як в прикладі на наступному слайді.

Інтерфейси

У цьому випадку клас Book реалізує інтерфейс Printable. При цьому треба враховувати, що якщо клас застосовує інтерфейс, то він повинен реалізувати всі методи інтерфейсу, як у випадку, наведеному вище, реалізовано метод print. Потім у методі main ми можемо створити об'єкт класу Book і викликати його метод print. Якщо клас не реалізує якісь методи інтерфейсу, то такий клас має бути визначений як абстрактний, а його неабстрактні класи-спадкоємці потім повинні будуть реалізувати ці методи.

```
public class Program{
    public static void main(String[] args) {
        Book firstBook = new Book( name: "Java. Complete Reference.", author: "H. Shildt");
        firstBook.print();
    }
}

1 usage 1 implementation
interface Printable{
    1 implementation
    void print();
}

2 usages
class Book implements Printable{
    2 usages
    String name;
    2 usages
    String author;
    1 usage
    Book(String name, String author){
        this.name = name;
        this.author = author;
    }

    public void print() {
        System.out.printf("%s (%s) \n", name, author);
    }
}
```

Інтерфейси

Водночас ми не можемо безпосередньо створювати об'єкти інтерфейсів, тому наступний код не працюватиме:

```
Printable pr = new Printable();  
pr.print();
```

Однією з переваг використання інтерфейсів є те, що вони дають змогу додати в додаток гнучкості. Наприклад, на додачу до класу **Book** визначимо ще один клас, який реалізовуватиме інтерфейс **Printable**.

Клас **Book** і клас **Journal** пов'язані тим, що вони реалізують інтерфейс **Printable**. Тому ми динамічно в програмі можемо створювати об'єкти **Printable** як екземпляри обох класів.

```
interface Printable{  
    1 implementation  
    void print();  
}  
  
no usages  
  
class Journal implements Printable {  
    3 usages  
    private String name;  
    no usages  
    String getName(){return name;}  
    no usages  
    Journal(String name){  
        | this.name = name;  
    }  
    public void print() {  
        | System.out.println(name);  
    }  
}
```

Интерфейси

```
public class Program{
    public static void main(String[] args) {
        Printable printable = new Book( name: "Java. Complete Reference", author: "H. Schildt");
        printable.print();    // Java. Complete Reference (H. Schildt)
        printable = new Journal( name: "Foreign Policy");
        printable.print();    // Foreign Policy
    }
}
3 usages 1 implementation
interface Printable{
    1 implementation
    void print();
}
1 usage
class Book implements Printable{
    2 usages
    String name;
    2 usages
    String author;
    1 usage
    Book(String name, String author){
        this.name = name;
        this.author = author;
    }

    public void print() {
        System.out.printf("%s (%s) \n", name, author);
    }
}
1 usage
class Journal implements Printable {
    3 usages
    private String name;
    no usages
    String getName(){return name;}
    1 usage
    Journal(String name){
        this.name = name;
    }
    public void print() {
        System.out.println(name);
    }
}
```

Інтерфейси

Інтерфейси в перетвореннях типів

Усе сказане щодо перетворення типів характерно і для інтерфейсів. Наприклад, оскільки клас `Journal` реалізує інтерфейс `Printable`, то змінна типу `Printable` може зберігати посилання на об'єкт типу `Journal`:

```
Printable p = new Journal( name: "Foreign Affairs");
p.print();
// Інтерфейс не має методу getName, необхідне явне приведення
String name = ((Journal)p).getName();
System.out.println(name);
```

І якщо ми хочемо звернутися до методів класу **Journal**, які визначені не в інтерфейсі **Printable**, а в самому класі **Journal**, то нам треба явно виконати перетворення типів: **((Journal)p).getName();**

Інтерфейси

Методи за замовчуванням

Раніше до **JDK 8** при реалізації інтерфейсу ми повинні були обов'язково реалізувати всі його методи в класі. А сам інтерфейс міг містити тільки визначення методів без конкретної реалізації. У **JDK 8** було додано таку функціональність як методи за замовчуванням. І тепер інтерфейси, крім визначення методів, можуть мати їхню реалізацію за замовчуванням, яка використовується, якщо клас, що реалізує цей інтерфейс, не реалізує метод. Наприклад, створимо метод за замовчуванням в інтерфейсі **Printable**:

```
interface Printable {  
    @Override  
    default void print(){  
        System.out.println("Undefined printable");  
    }  
}
```

Інтерфейси

Метод за замовчуванням - це звичайний метод без модифікаторів, який позначається ключовим словом default. Потім у класі Journal нам необов'язково цей метод реалізувати, хоча ми можемо його і перевизначити.

```
class Journal implements Printable {  
    2 usages  
    private String name;  
    1 usage  
    String getName(){  
        return name;  
    }  
    1 usage  
    Journal(String name){  
        this.name = name;  
    }  
}
```

Інтерфейси

Статичні методи

Починаючи з JDK 8 в інтерфейсах доступні статичні методи - вони аналогічні методам класу:

```
interface Printable {  
    // implementation  
    void print();  
    // no usages  
    static void read(){  
        System.out.println("Read printable");  
    }  
}
```

Щоб звернутися до статичного методу інтерфейсу також, як і у випадку з класами, пишуть назву інтерфейсу і метод:

Printable.read();

Інтерфейси

Приватні методи

За замовчуванням усі методи в інтерфейсі фактично мають модифікатор **public**. Однак починаючи з **Java 9** ми також можемо визначати в інтерфейсі методи з модифікатором `private`. Вони можуть бути статичними і нестатичними, але вони не можуть мати реалізації за замовчуванням. Подібні методи можуть використовуватися тільки всередині самого інтерфейсу, в якому вони визначені. Тобто, наприклад, нам треба виконувати в інтерфейсі деякі дії, що повторюються, і в цьому випадку такі дії можна виділити в приватні методи.

```
public class Program{
    public static void main(String[] args) {
        Calculatable c = new Calculation();
        System.out.println(c.sum( a: 1, b: 2));
        System.out.println(c.sum( a: 1, b: 2, c: 4));
    }
}

1 usage
class Calculation implements Calculatable{ }

2 usages 1 implementation
interface Calculatable{
    1 usage
    default int sum(int a, int b){
        return sumAll(a, b);
    }

    1 usage
    default int sum(int a, int b, int c){
        return sumAll(a, b, c);
    }

    2 usages
    private int sumAll(int... values){
        int result = 0;
        for(int n : values){
            result += n;
        }
        return result;
    }
}
```

Інтерфейси

Константи в інтерфейсах

Крім методів в інтерфейсах можуть бути визначені статичні константи.

```
interface Stateable{  
    no usages  
    int OPEN = 1;  
    no usages  
    int CLOSED = 0;  
  
    no usages  
    void printState(int n);  
}
```

Хоча такі константи також не мають модифікаторів, але за замовчуванням вони мають модифікатор доступу `public static final`, і тому їхнє значення доступне з будь-якого місця програми. На наступному слайді приведено приклад застосування констант.

Интерфейси

```
public class Program{
    public static void main(String[] args) {
        WaterPipe pipe = new WaterPipe();
        pipe.printState( n: 1);
    }
}

2 usages
class WaterPipe implements Stateable{
    1 usage
    public void printState(int n){
        if(n==OPEN)
            System.out.println("Water is opened");
        else if(n==CLOSED)
            System.out.println("Water is closed");
        else
            System.out.println("State is invalid");
    }
}

1 usage 1 implementation
interface Stateable{
    1 usage
    int OPEN = 1;
    1 usage
    int CLOSED = 0;
    1 usage
    void printState(int n);
}
```

Інтерфейси

Множинна реалізація інтерфейсів

Якщо нам треба застосувати в класі кілька інтерфейсів, то вони всі перераховуються через кому після слова implements:

```
1usage
interface Printable {
    // методи інтерфейсу
}

1usage 1implementation
interface Searchable {
    // методи інтерфейсу
}

no usages
class Book implements Printable, Searchable{
    // реалізація класу
}
```

Інтерфейси

Успадкування інтерфейсів

Інтерфейси, як і класи, можуть успадковуватися:

```
interface BookPrintable extends Printable{  
    no usages  
    void paint();  
}
```

При застосуванні цього інтерфейсу клас Book повинен буде реалізувати як методи інтерфейсу BookPrintable, так і методи базового інтерфейсу Printable.

Вкладені інтерфейси

Як і класи, інтерфейси можуть бути вкладеними, тобто можуть бути визначені в класах або інших інтерфейсах. Наприклад:

```
class Printer{  
    no usages  
    interface Printable {  
        void print();  
    }  
}
```

Інтерфейси

При застосуванні такого інтерфейсу нам треба вказувати його повне ім'я разом з ім'ям класу:

```
class Journal implements Printer.Printable {  
    2 usages  
    String name;  
    no usages  
    Journal(String name){  
        this.name = name;  
    }  
    public void print() {  
        System.out.println(name);  
    }  
}
```

Використання інтерфейсу буде аналогічним до попередніх випадків:

```
Printer.Printable p = new Journal( name: "Foreign Affairs");  
p.print();
```

Інтерфейси

Інтерфейси як параметри та результати методів

І так само як і у випадку з класами, інтерфейси можуть використовуватися як тип параметрів методу або як тип, що повертається.

Метод **read()** як параметр приймає об'єкт інтерфейсу **Printable**, тому в цей метод ми можемо передати як об'єкт **Book**, так і об'єкт **Journal**.

Метод **createPrintable()** повертає об'єкт **Printable**, тому також ми можемо повернути як об'єкт **Book**, так і **Journal**.

```
public class Program{
    public static void main(String[] args) {
        Printable printable = createPrintable( name: "Foreign Affairs", option: false);
        printable.print();
        read(new Book( name: "Java for impatient", author: "Cay Horstmann"));
        read(new Journal( name: "Java Dayly News"));
    }

    2 usages
    static void read(Printable p){ p.print();}

    1 usage
    static Printable createPrintable(String name, boolean option){
        if(option)
            return new Book(name, author: "Undefined");
        else
            return new Journal(name);
    }
}

5 usages 2 implementations
interface Printable{
    2 implementations
    void print();
}

2 usages
class Book implements Printable{
    2 usages
    String name;
    2 usages
    String author;
    2 usages
    Book(String name, String author){
        this.name = name;
        this.author = author;
    }
    public void print() { System.out.printf("%s (%s) \n", name, author);}
}

2 usages
class Journal implements Printable {
    3 usages
    private String name;
    no usages
    String getName(){ return name;}
    2 usages
    Journal(String name){ this.name = name;}
    public void print() { System.out.println(name); }
}
```

Інтерфейси в механізмі зворотного виклику

Одним із поширених способів використання інтерфейсів у Java є створення **зворотного виклику**. Суть зворотного виклику полягає в тому, що ми створюємо дії, які викликаються при інших діях. Тобто одні дії викликаються іншими діями. Стандартний приклад - натискання на кнопку. Коли ми натискаємо на кнопку, ми виконуємо дію, але у відповідь на це натискання запускаються інші дії. Наприклад, натискання на значок принтера запускає друк документа на принтері тощо. Розглянемо приклад на наступному слайді.

```
public class Program {
    public static void main(String[] args) {
        Button button = new Button(new ButtonClickHandler());
        button.click();
        button.click();
        button.click();
    }
}

1 usage
class ButtonClickHandler implements EventHandler{
    public void execute(){
        System.out.println("Кнопка натиснута!");
    }
}

3 usages 1 implementation
interface EventHandler{
    1 implementation
    void execute();
}

2 usages
class Button{
    2 usages
    EventHandler handler;
    1 usage
    Button(EventHandler action){
        this.handler=action;
    }
    3 usages
    public void click(){
        handler.execute();
    }
}
```

Отже, тут у нас визначено клас **Button**, який у конструкторі приймає об'єкт інтерфейсу **EventHandler** і в методі `click` (імітація натискання) викликає метод `execute` цього об'єкта. Далі визначається реалізація **EventHandler** у вигляді класу **ButtonClickHandler**. І в основній програмі об'єкт цього класу передається в конструктор **Button**. Таким чином, через конструктор ми встановлюємо обробник натискання кнопки. І при кожному виклику методу **button.click()** буде викликатися цей обробник.

Інтерфейси в механізмі зворотного виклику

Але здавалося б, навіщо нам виносити всі дії в інтерфейс, його реалізувати, чому б не написати простіше відразу в класі Button:

```
class Button{  
    3 usages  
    public void click(){  
        System.out.println("Кнопка натиснута!");  
    }  
}
```

Річ у тім, що на момент визначення класу нам не завжди бувають точно відомі ті дії, які мають виконуватися. Особливо якщо клас Button і клас основної програми знаходяться в різних пакетах, бібліотеках і можуть проектуватися різними розробниками. До того ж у нас може бути кілька кнопок - об'єктів Button і для кожного об'єкта треба визначити свою дію. Наприклад, змінимо головний клас програми на наступному слайді.

Інтерфейси в механізмі зворотного виклику

```
public class Program {  
  
    public static void main(String[] args) {  
  
        Button tvButton = new Button(new EventHandler(){  
            3 usages  
            private boolean on = false;  
            public void execute(){  
  
                if(on) {  
                    System.out.println("Телевізор вимкнений..");  
                    on=false;  
                }  
                else {  
                    System.out.println("Телевізор увімкнений!");  
                    on=true;  
                }  
            }  
        });  
        Button printButton = new Button(() -> System.out.println("Запущено друк на принтері..."));  
  
        tvButton.click();  
        printButton.click();  
        tvButton.click();  
    }  
}
```

Тут у нас дві кнопки - одна для увімкнення-вимкнення телевізора, а інша для друку на принтері. Замість того, щоб створювати окремі класи, що реалізують інтерфейс **EventHandler**, тут обробники задаються у вигляді анонімних об'єктів, які реалізують інтерфейс **EventHandler**. Причому обробник кнопки телевізора зберігає додатковий стан у вигляді логічної змінної `on`.

І на завершення треба сказати, що інтерфейси в даній якості особливо широко використовуються в різних графічних API - **AWT**, **Swing**, **JavaFX**, де обробка подій об'єктів - елементів графічного інтерфейсу особливо актуальна.

Інтерфейси в механізмі зворотного виклику

З точки зору **об'єктно-орієнтованого програмування**, основна відмінність між інтерфейсом та абстрактним класом полягає в тому, що інтерфейс не може мати стан, тоді як абстрактний клас може мати стан зі змінними екземпляра.

Інша ключова відмінність полягає в тому, що класи можуть реалізовувати більше одного **інтерфейсу**, але вони можуть розширювати тільки один **абстрактний** клас. Це дизайнерське рішення ґрунтується на тому, що множинне успадкування (розширення більш ніж одного класу) може призвести до тупикових ситуацій у коді.

Інженери Java вирішили уникнути цього. Ще одна відмінність полягає в тому, що інтерфейси можуть бути реалізовані класами або розширені інтерфейсами, але класи можуть бути тільки розширені.

Важливо також зазначити, що **лямбда-вирази можна використовувати лише з функціональним інтерфейсом** (тобто інтерфейсом з одним методом), тоді як абстрактні класи з одним абстрактним методом не можуть використовувати лямбди.

Інтерфейси в механізмі зворотного виклику

Parameters	Interface	Abstract class
Speed	Slow	Fast
Multiple Inheritances	Implement several Interfaces	Only one abstract class
Structure	Abstract methods	Abstract & concrete methods
When to use	Future enhancement	To avoid independence
Inheritance/ Implementation	A Class can implement multiple interfaces	The class can inherit only one Abstract Class
Default Implementation	While adding new stuff to the interface, it is a nightmare to find all the implementors and implement newly defined stuff.	In case of Abstract Class, you can take advantage of the default implementation.
Access Modifiers	The interface does not have access modifiers. Everything defined inside the interface is assumed public modifier.	Abstract Class can have an access modifier.
When to use	It is better to use interface when various implementations share only method signature. Polymorphic hierarchy of value types.	It should be used when various implementations of the same kind share a common behavior.
Data fields	the interface cannot contain data fields.	the class can have data fields.
Multiple Inheritance Default	A class may implement numerous interfaces.	A class inherits only one abstract class.
Implementation	An interface is abstract so that it can't provide any code.	An abstract class can give complete, default code which should be overridden.
Use of Access modifiers	You cannot use access modifiers for the method, properties, etc.	You can use an abstract class which contains access modifiers.
Usage	Interfaces help to define the peripheral abilities of a class.	An abstract class defines the identity of a class.
Defined fields	No fields can be defined	An abstract class allows you to define both fields and constants
Inheritance	An interface can inherit multiple interfaces but cannot inherit a class.	An abstract class can inherit a class and multiple interfaces.
Constructor or destructors	An interface cannot declare constructors or destructors.	An abstract class can declare constructors and destructors.
Limit of Extensions	It can extend any number of interfaces.	It can extend only one class or one abstract class at a time.
Abstract keyword	In an abstract interface keyword, is optional for declaring a method as an abstract.	In an abstract class, the abstract keyword is compulsory for declaring a method as an abstract.
Class type	An interface can have only public abstract methods.	An abstract class has protected and public abstract methods.

Перерахування enum

Крім окремих примітивних типів даних і класів у Java є такий тип як **enum** або **перерахування**. Перерахування представляють набір логічно пов'язаних констант. Оголошення перерахування відбувається за допомогою оператора `enum`, після якого йде назва перерахування. Потім йде список елементів перерахування через кому.

```
no usages
enum Day{
    no usages
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}
```

Перерахування фактично представляє новий тип, тому ми можемо визначити змінну цього типу і використовувати її:

Перерахування enum

```
public class Program{
    public static void main(String[] args) {
        Day current = Day.MONDAY;
        System.out.println(current);    // MONDAY
    }
}

2 usages
enum Day{
    1 usage
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}
```

Перерахування можуть використовуватися в класах для зберігання даних:

```
public class Program{
    public static void main(String[] args) {
        Book b1 = new Book( name: "War and Peace", author: "L. Tolstoy", Type.BELLETRE);
        System.out.printf("Book '%s' has a type %s", b1.name, b1.bookType);
        switch(b1.bookType){
            case BELLETRE:
                System.out.println("Belletre");
                break;
            case SCIENCE:
                System.out.println("Science");
                break;
            case SCIENCE_FICTION:
                System.out.println("Science fiction");
                break;
            case PHANTASY:
                System.out.println("Phantasy");
                break;
        }
    }
}

2 usages
class Book{
    2 usages
    String name;
    3 usages
    Type bookType;
    1 usage
    String author;
    1 usage
    Book(String name, String author, Type type){
        bookType = type;
        this.name = name;
        this.author = author;
    }
}

3 usages
enum Type
{
    1 usage
    SCIENCE, BELLETRE, PHANTASY, SCIENCE_FICTION
}
```

Перерахування enum

Саме перерахування оголошено поза класом, воно містить чотири жанри книг. Клас **Book** крім звичайних змінних містить також змінну типу нашого перерахування. У конструкторі ми її також можемо присвоїти, як і звичайні поля класу.

За допомогою конструкції **switch..case** можна перевірити приналежність значення bookType певній константі перерахування.

Методи перерахувань

Кожне перерахування має статичний метод values(). Він повертає масив усіх констант перерахування. Метод ordinal() повертає порядковий номер певної константи (нумерація починається з 0):

```
public class Program{
    public static void main(String[] args) {
        Type[] types = Type.values();
        for (Type s : types) { System.out.println(s);
        }
    }
}
3 usages
enum Type {
    no usages
    SCIENCE, BELLETRE, PHANTASY, SCIENCE_FICTION
}
```

Перерахування enum

Конструктори, поля та методи перерахування

Перерахування, як і звичайні класи, можуть визначати конструктори, поля та методи. Наприклад:

```
public class Program{
    public static void main(String[] args) {
        System.out.println(Color.RED.getCode());           // #FF0000
        System.out.println(Color.GREEN.getCode());         // #00FF00
    }
}

2 usages
enum Color{
    1 usage
    RED( code: "#FF0000"), BLUE( code: "#0000FF"), GREEN( code: "#00FF00");
    2 usages
    private String code;
    3 usages
    Color(String code){ this.code = code;}
    2 usages
    public String getCode(){ return code;}
}
```


Перерахування enum

Перерахування **Color** визначає приватне поле `code` для зберігання коду кольору, а за допомогою методу **getCode** воно повертається. Через конструктор передається для нього значення. Слід зазначити, що конструктор за замовчуванням приватний, тобто має модифікатор `private`.

Будь-який інший модифікатор вважатиметься помилкою. Тому створити константи перерахування за допомогою конструктора ми можемо тільки всередині перерахування. Також можна визначати методи для окремих констант.

```
public class Program{
    public static void main(String[] args) {
        Operation op = Operation.SUM;
        System.out.println(op.action(x: 10, y: 4)); // 14
        op = Operation.MULTIPLY;
        System.out.println(op.action(x: 6, y: 4)); // 24
    }
}
3 usages 3 inheritors
enum Operation{
    1 usage
    SUM{
        2 usages
        public int action(int x, int y){ return x + y;}
    },
    no usages
    SUBTRACT{
        2 usages
        public int action(int x, int y){ return x - y;}
    },
    1 usage
    MULTIPLY{
        2 usages
        public int action(int x, int y){ return x * y;}
    };
    2 usages 3 implementations
    public abstract int action(int x, int y);
}
```

Узагальнення (Generics)

Узагальнення або **generics** (узагальнені типи і методи) дають нам змогу уникнути жорсткого визначення використовуваних типів. Розглянемо проблему, у якій вони нам можуть знадобитися. Припустимо, ми визначаємо клас для представлення банківського рахунку. Наприклад, він міг би мати такий вигляд:

```
class Account{  
    2 usages  
    private int id;  
    3 usages  
    private int sum;  
  
    no usages  
    Account(int id, int sum){  
        this.id = id;  
        this.sum = sum;  
    }  
  
    no usages  
    public int getId() { return id; }  
    no usages  
    public int getSum() { return sum; }  
    no usages  
    public void setSum(int sum) { this.sum = sum; }  
}
```

Узагальнення (Generics)

Клас **Account** має два поля: `id` - унікальний ідентифікатор рахунку і `sum` - сума на рахунку.

У цьому разі ідентифікатор задано як цілочисельне значення, наприклад, 1, 2, 3, 4 і так далі. Однак також нерідко для ідентифікатора використовують і рядкові значення. І числові, і рядкові значення мають свої плюси і мінуси. І на момент написання класу ми можемо точно не знати, що краще вибрати для зберігання ідентифікатора - рядки або числа. Або, можливо, цей клас використовуватимуть інші розробники, які можуть мати свою думку щодо цієї проблеми. Наприклад, як тип `id` вони захочуть використовувати якийсь свій клас.

І на перший погляд ми можемо розв'язати цю проблему таким чином: задати `id` як поле типу `Object`, який є універсальним і базовим суперкласом для всіх інших типів.

Узагальнення (Generics)

```
public class Program{
    public static void main(String[] args) {
        Account acc1 = new Account( id: 2334, sum: 5000); // id - число
        int acc1Id = (int)acc1.getId();
        System.out.println(acc1Id);

        Account acc2 = new Account( id: "sid5523", sum: 5000); // id - строка
        System.out.println(acc2.getId());
    }
}
4 usages
class Account{
    2 usages
    private Object id;
    3 usages
    private int sum;
    2 usages
    Account(Object id, int sum){
        this.id = id;
        this.sum = sum;
    }
    2 usages
    public Object getId() { return id; }
    no usages
    public int getSum() { return sum; }
    no usages
    public void setSum(int sum) { this.sum = sum; }
}
```

У цьому випадку все чудово працює. Однак тоді ми стикаємося з проблемою безпеки типів. Наприклад, у такому випадку ми отримаємо помилку:

```
Account acc1 = new Account( id: "2345", sum: 5000);
int acc1Id = (int)acc1.getId(); // java.lang.ClassCastException
System.out.println(acc1Id);
```

Узагальнення (Generics)

Проблема може здатися штучною, оскільки в даному випадку ми бачимо, що в конструктор передається рядок, тому ми навряд чи будемо намагатися перетворювати його до типу `int`. Однак у процесі розробки ми можемо не знати, який саме тип представляє значення в `id`, і під час спроби отримати число в цьому разі ми зіткнемося з виключенням ***java.lang.ClassCastException***.

Писати для кожного окремого типу свою версію класу **Account** теж не є гарним рішенням, оскільки в цьому випадку ми змушені повторюватися. Ці проблеми були покликані усунути узагальнення або generics. Узагальнення дозволяють не вказувати конкретний тип, який буде використовуватися. Тому визначимо клас **Account** як узагальнений (на наступному слайді).

Узагальнення (Generics)

За допомогою літери **T** у визначенні класу `class Account<T>` ми вказуємо, що даний тип **T** буде використовуватися цим класом. Параметр **T** у кутових дужках називається універсальним параметром, оскільки замість нього можна підставити будь-який тип. При цьому поки ми не знаємо, який саме це буде тип: `String`, `int` або якийсь інший. Причому буква **T** обрана умовно, це може і будь-яка інша буква або набір символів.

```
class Account<T>{  
  
    2 usages  
    private T id;  
    3 usages  
    private int sum;  
  
    no usages  
    Account(T id, int sum){  
        this.id = id;  
        this.sum = sum;  
    }  
  
    no usages  
    public T getId() { return id; }  
    no usages  
    public int getSum() { return sum; }  
    no usages  
    public void setSum(int sum) { this.sum = sum; }  
}
```

Після оголошення класу ми можемо застосувати універсальний параметр **T**: так далі в класі оголошується змінна цього типу, якій потім присвоюється значення в конструкторі.

Метод **getId()** повертає значення змінної **id**, але оскільки ця змінна представляє тип **T**, то цей метод також повертає об'єкт типу **T**: **public T getId()**.

Узагальнення (Generics)

```
public class Program{
    public static void main(String[] args) {
        Account<String> acc1 = new Account<String>( id: "2345", sum: 5000);
        String acc1Id = acc1.getId();
        System.out.println(acc1Id);

        Account<Integer> acc2 = new Account<Integer>( id: 2345, sum: 5000);
        Integer acc2Id = acc2.getId();
        System.out.println(acc2Id);
    }
}
4 usages
class Account<T>{
    2 usages
    private T id;
    3 usages
    private int sum;
    2 usages
    Account(T id, int sum){
        this.id = id;
        this.sum = sum;
    }
    2 usages
    public T getId() { return id; }
    no usages
    public int getSum() { return sum; }
    no usages
    public void setSum(int sum) { this.sum = sum; }
}
```

При визначенні змінної даного класу і створенні об'єкта після імені класу в кутових дужках потрібно вказати, який саме тип буде використовуватися замість універсального параметра. При цьому треба враховувати, що вони працюють тільки з об'єктами, але не працюють із примітивними типами.

Тобто ми можемо написати **Account<Integer>**, але не можемо використовувати тип `int` або `double`, наприклад, **Account<int>**. Замість примітивних типів треба використовувати класи-обгортки: `Integer` замість **int**, **Double** замість **double** тощо.

Узагальнення (Generics)

Узагальнені інтерфейси

Інтерфейси, як і класи, також можуть бути узагальненими. Створимо узагальнений інтерфейс **Accountable** і використаємо його в програмі:

```
public class Program{
    public static void main(String[] args) {
        Accountable<String> acc1 = new Account( id: "1235rwr", sum: 5000);
        Account acc2 = new Account( id: "2373", sum: 4300);
        System.out.println(acc1.getId());
        System.out.println(acc2.getId());
    }
}

2 usages 1 implementation
interface Accountable<T>{
    2 usages 1 implementation
    T getId();
    no usages 1 implementation
    int getSum();
    no usages 1 implementation
    void setSum(int sum);
}

3 usages
class Account implements Accountable<String>{
    2 usages
    private String id;
    3 usages
    private int sum;

    2 usages
    Account(String id, int sum){
        this.id = id;
        this.sum = sum;
    }

    2 usages
    public String getId() { return id; }
    no usages
    public int getSum() { return sum; }
    no usages
    public void setSum(int sum) { this.sum = sum; }
}
```


Узагальнення (Generics)

Під час реалізації подібного інтерфейсу є дві стратегії. У цьому випадку реалізовано першу стратегію, коли під час реалізації для універсального параметра інтерфейсу задають конкретний тип, як, наприклад, у цьому випадку це тип **String**. Тоді клас, що реалізує інтерфейс, жорстко прив'язаний до цього типу.

Друга стратегія представляє визначення узагальненого класу, який також використовує той самий універсальний параметр:

```
public class Program {
    public static void main(String[] args) {
        Account<String> acc1 = new Account<String>(id: "1235rwr", sum: 5000);
        Account<String> acc2 = new Account<String>(id: "2373", sum: 4300);
        System.out.println(acc1.getId());
        System.out.println(acc2.getId());
    }
}

1 usage 1 implementation
interface Accountable<T>{
    2 usages 1 implementation
    T getId();
    no usages 1 implementation
    int getSum();
    no usages 1 implementation
    void setSum(int sum);
}

4 usages
class Account<T> implements Accountable<T>{
    2 usages
    private T id;
    3 usages
    private int sum;
    2 usages
    Account(T id, int sum){
        this.id = id;
        this.sum = sum;
    }
    2 usages
    public T getId() { return id; }
    no usages
    public int getSum() { return sum; }
    no usages
}
```

Узагальнення (Generics)

Узагальнені методи

Крім узагальнених типів можна також створювати узагальнені методи, які так само будуть використовувати універсальні параметри. Наприклад:

```
public class Program{
    public static void main(String[] args) {
        Printer printer = new Printer();
        String[] people = {"Tom", "Alice", "Sam", "Kate", "Bob", "Helen"};
        Integer[] numbers = {23, 4, 5, 2, 13, 456, 4};
        printer.<String>print(people);
        printer.<Integer>print(numbers);
    }
}

2 usages
class Printer{
    public <T> void print(T[] items){
        for(T item: items){
            System.out.println(item);
        }
    }
}
```

Узагальнення (Generics)

Особливістю узагальненого методу є використання **універсального параметра** в оголошенні методу після всіх модифікаторів і перед типом значення, що повертається.

```
public <T> void print(T[] items)
```

Потім усередині методу всі значення типу **T** представлятимуть цей універсальний параметр. Під час виклику подібного методу перед його ім'ям у кутових дужках вказується, який тип буде передаватися на місце універсального параметра:

```
printer.<String>print(people);  
printer.<Integer>print(numbers);
```

Узагальнення (Generics)

Використання декількох універсальних параметрів Ми можемо також задати відразу кілька універсальних параметрів:

```
public class Program{
    public static void main(String[] args) {
        Account<String, Double> acc1 = new Account<String, Double>(id: "354", sum: 5000.87);
        String id = acc1.getId();
        Double sum = acc1.getSum();
        System.out.printf("Id: %s Sum: %f \n", id, sum);
    }
}
2 usages
class Account<T, S>{
    2 usages
    private T id;
    3 usages
    private S sum;
    1 usage
    Account(T id, S sum){
        this.id = id;
        this.sum = sum;
    }
    1 usage
    public T getId() { return id; }
    1 usage
    public S getSum() { return sum; }
    no usages
    public void setSum(S sum) { this.sum = sum; }
}
```

У цьому випадку тип **String** передаватиметься на місце параметра **T**, а тип **Double** - на місце параметра **S**.

Узагальнення (Generics)

Узагальнені конструктори

Конструктори як і методи також можуть бути узагальненими. У цьому випадку перед конструктором також вказуються в кутових дужках універсальні параметри.

У цьому випадку конструктор приймає параметр `id`, який представляє тип `T`. У конструкторі його значення перетворюється на рядок і зберігається в локальну змінну.

```
public class Program1
{
    public static void main(String[] args) {
        Account acc1 = new Account( id: "cid2373", sum: 5000);
        Account acc2 = new Account( id: 53757, sum: 4000);
        System.out.println(acc1.getId());
        System.out.println(acc2.getId());
    }
}

4 usages
class Account{
    2 usages
    private String id;
    3 usages
    private int sum;
    2 usages
    <T>Account(T id, int sum){
        this.id = id.toString();
        this.sum = sum;
    }
    2 usages
    public String getId() { return id; }
    no usages
    public int getSum() { return sum; }
    no usages
    public void setSum(int sum) { this.sum = sum; }
}
```

Обмеження узагальнень

Коли ми вказуємо універсальний параметр узагальнень, то за замовчуванням він може представляти **будь-який тип**. Однак іноді необхідно, щоб параметр відповідав тільки деякому обмеженому набору типів. У цьому разі застосовуються обмеження, які дають змогу вказати базовий клас, якому має відповідати параметр. Для встановлення обмеження після універсального параметра ставиться слово **extends**, після якого вказується базовий клас обмеження:

```
class Account{ }  
class Transaction<T extends Account>{ }
```

Наприклад, у цьому випадку для параметра **T** у **Transaction** обмеженням є клас **Account**. Тобто на місце параметра **T** ми можемо передати або клас **Account**, або один із його класів-спадкоємців. Наприклад, розглянемо таку програму на наступному слайді.

Обмеження узагальнень

```
public class Program1
{
    public static void main(String[] args) {
        Account acc1 = new Account( id: "1876", sum: 4500);
        Account acc2 = new Account( id: "3476", sum: 1500);
        Transaction<Account> tran1 = new Transaction<Account>(acc1, acc2, sum: 4000);
        tran1.execute();
        tran1 = new Transaction<Account>(acc1, acc2, sum: 4000);
        tran1.execute();
    }
}
3 usages
class Transaction<T extends Account>{
    6 usages
    private T from;
    5 usages
    private T to;
    4 usages
    private int sum;
    2 usages
    Transaction(T from, T to, int sum){
        this.from = from;
        this.to = to;
        this.sum = sum;
    }
    public void execute(){
        if (from.getSum() > sum) { from.setSum(from.getSum() - sum); to.setSum(to.getSum() + sum);
            System.out.printf("Account %s: %d \nAccount %s: %d \n", from.getId(), from.getSum(), to.getId(), to.getSum());
        }
        else System.out.printf("Operation is invalid");
    }
}
```

```
class Account{
    2 usages
    private String id;
    3 usages
    private int sum;
    2 usages
    Account(String id, int sum){
        this.id = id;
        this.sum = sum;
    }
    2 usages
    public String getId() { return id; }
    5 usages
    public int getSum() { return sum; }
    2 usages
    public void setSum(int sum) { this.sum = sum; }
}
```

Обмеження узагальнень

У цьому випадку клас **Transaction**, який представляє операцію переказу коштів між двома рахунками, типізовано параметром **T**, у якого як обмеження встановлено клас **Account**. При створенні об'єкта **Transaction** в його конструктор передаються два об'єкти **Account** - два рахунки, між якими треба здійснити переказ, і сума переказу.

При цьому важливо розуміти, що оскільки ми встановили подібне обмеження, то компілятор буде розпізнавати об'єкти типу **T** як об'єкти типу **Account**. І в цьому випадку ми можемо викликати в об'єктів типу **T** методи класу **Account**. І ми б не змогли б це зробити, якби ми не поставили подібного обмеження:

```
class Transaction<T>{  
    // решта вмісту  
}
```

У цьому випадку була б помилка.

Обмеження узагальнень

Узагальнені типи як обмеження

Як обмеження можуть виступати й інші узагальнення, які самі можуть мати обмеження.

У цьому випадку обмеженням для Transaction є тип Account, який типізовано типом String.

```
public class Program{

    public static void main(String[] args) {

        Account<String> acc1 = new Account<String>(id: "1876", sum: 4500);
        Account<String> acc2 = new Account<String>(id: "3476", sum: 1500);

        Transaction<Account<String>> tran1 = new Transaction<Account<String>>(acc1, acc2, sum: 4000);
        tran1.execute();
        tran1 = new Transaction<Account<String>>(acc1, acc2, sum: 4000);
        tran1.execute();
    }
}

3 usages
class Transaction<T extends Account<String>>{

    6 usages
    private T from; // с какого счета перевод
    5 usages
    private T to; // на какой счет перевод
    4 usages
    private int sum; // сумма перевода

    2 usages
    Transaction(T from, T to, int sum){
        this.from = from;
        this.to = to;
        this.sum = sum;
    }

    public void execute(){

        if (from.getSum() > sum)
        {
            from.setSum(from.getSum() - sum);
            to.setSum(to.getSum() + sum);
            System.out.printf("Account %s: %d \nAccount %s: %d \n",
                               from.getId(), from.getSum(), to.getId(), to.getSum());
        }
        else{
            System.out.printf("Operation is invalid");
        }
    }
}
```

Обмеження узагальнень

Інтерфейси обмеження

Як обмеження можуть виступати також інтерфейси. У цьому разі тип, що передається на місце універсального параметра, повинен реалізувати цей інтерфейс:

```
public class Program {
    public static void main(String[] args) {
        Account acc1 = new Account( id: "1235rwr", sum: 5000);
        Account acc2 = new Account( id: "2373", sum: 4300);
        Transaction<Account> tran1 = new Transaction<Account>(acc1, acc2, sum: 1560);
        tran1.execute();
    }
}

2 usages 1 implementation
interface Accountable {
    2 usages 1 implementation
    String getId();
    5 usages 1 implementation
    int getSum();
    2 usages 1 implementation
    void setSum(int sum);
}

6 usages
class Account implements Accountable {
    2 usages
    private String id;
    3 usages
    private int sum;
    2 usages
    Account(String id, int sum) {
        this.id = id;
        this.sum = sum;
    }
    2 usages
    public String getId() { return id; }
    5 usages
    public int getSum() { return sum; }
    2 usages
    public void setSum(int sum) { this.sum = sum; }
}
```

```
class Transaction<T extends Accountable> {
    6 usages
    private T from; // с какого счета перевод
    5 usages
    private T to; // на какой счет перевод
    4 usages
    private int sum; // сумма перевода
    1 usage
    Transaction(T from, T to, int sum) {
        this.from = from;
        this.to = to;
        this.sum = sum;
    }
    public void execute() {
        if (from.getSum() > sum) {
            from.setSum(from.getSum() - sum);
            to.setSum(to.getSum() + sum);
            System.out.printf("Account %s: %d \nAccount %s: %d \n",
                               from.getId(), from.getSum(), to.getId(), to.getSum());
        }
        else System.out.printf("Operation is invalid");
    }
}
```

Обмеження узагальнень

Множинні обмеження

Також можна встановити одразу кілька обмежень. Наприклад, нехай клас Transaction може працювати тільки з об'єктами, які одночасно реалізують інтерфейс Accountable і є спадкоємцями класу Person:

```
class Person{}  
  
1 usage  
interface Accountable{}  
  
no usages  
class Transaction<T extends Person & Accountable>{}
```

Контрольні питання

1. Надайте визначення абстрактним класам у контексті об'єктно-орієнтованого програмування.
2. Надайте визначення інтерфейсам та зазначте їх роль у визначенні контрактів для класів у Java.
3. Охарактеризуйте перерахування (enum) як спеціальний тип в мові програмування Java та поясніть його призначення для визначення константних значень.
4. Зазначте мету конструкторів у класі та поясніть, для чого вони використовуються в програмуванні на Java.
5. Охарактеризуйте узагальнення (generics) в мові програмування Java та поясніть його роль у створенні загальноприйнятих алгоритмів та структур даних.
6. Розкрийте різницю між абстрактними класами та інтерфейсами в контексті мови програмування Java, визначаючи їхні основні властивості та використання.

The END
Lec 6.

.NET та JAVA #7

Обробка рядків

1. Введення в рядки. Клас String
2. Основні операції з рядками
3. StringBuffer та StringBuilder
4. Регулярні вирази

Введення в рядки. Клас String

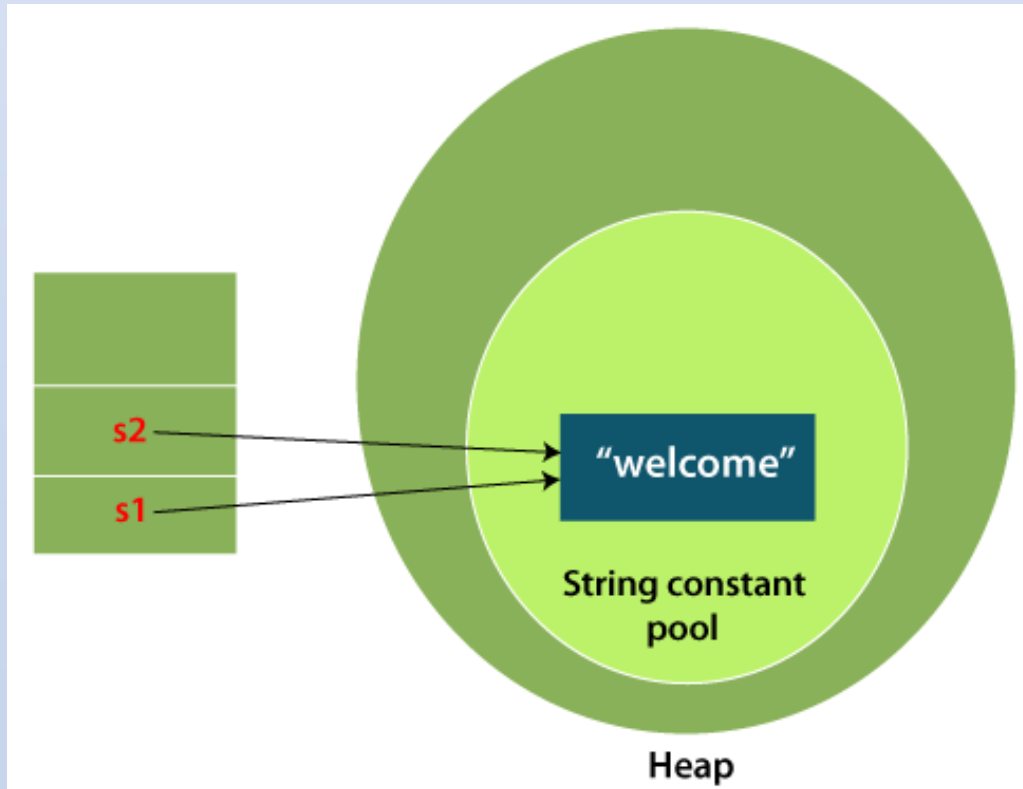
Рядок (String) являє собою послідовність символів. Для роботи з рядками в Java визначено клас **String**, який надає низку методів для маніпуляції рядками. Фізично об'єкт **String** являє собою посилання на ділянку в пам'яті, в якій розміщені символи.

Для створення нового рядка ми можемо використати один із конструкторів класу **String**, або безпосередньо присвоїти рядок у подвійних лапках.

```
public static void main(String[] args) {  
  
    String str1 = "Java";  
    String str2 = new String(); // пустий рядок  
    String str3 = new String(new char[] {'h', 'e', 'l', 'l', 'o'});  
    String str4 = new String(new char[]{'w', 'e', 'l', 'c', 'o', 'm', 'e'}, offset: 3, count: 4); //3 -початковий індекс, 4 - кількість символів  
  
    System.out.println(str1); // Java  
    System.out.println(str2); //  
    System.out.println(str3); // hello  
    System.out.println(str4); // come  
}
```

Введення в рядки. Клас String

Кожного разу, коли ви створюєте рядковий літерал, JVM спочатку перевіряє "пул рядкових констант" (**string constant pool**). Якщо рядок вже існує у пулі, повертається посилання на його екземпляр. Якщо рядка не існує у пулі, створюється новий екземпляр рядка і розміщується у пулі.

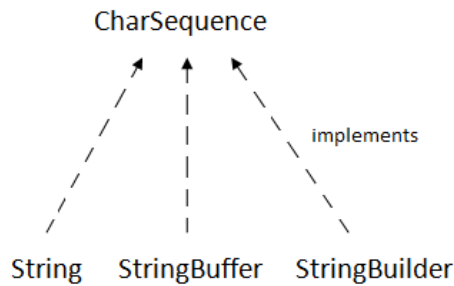


```
String s1="Welcome";  
String s2="Welcome";//Не створюється новий екземпляр
```


Введення в рядки. Клас String

Інтерфейс CharSequence

Використовується для представлення послідовності символів. Його реалізують класи **String**, **StringBuffer** та **StringBuilder**. Це означає, що ми можемо створювати рядки в Java за допомогою цих трьох класів.



Рядок **Java** є незмінним, тобто його не можна змінити. Кожного разу, коли ми змінюємо будь-який рядок, створюється новий екземпляр. Для змінних рядків можна використовувати класи **StringBuffer** та **StringBuilder**. Про незмінні рядки ми поговоримо пізніше. Спочатку давайте розберемося, що таке `String` в Java і як створити об'єкт `String`.

Введення в рядки. Клас String

Під час роботи з рядками важливо розуміти, що об'єкт **String** є **незмінним (immutable)**. Тобто при будь-яких операціях над рядком, які змінюють цей рядок, фактично буде створюватися новий рядок.

Оскільки рядок розглядається як набір символів, то ми можемо застосувати метод **length()** для знаходження довжини рядка або довжини набору символів.

```
String str1 = "Java";  
System.out.println(str1.length()); // 4
```

А за допомогою методу **toCharArray()** можна назад перетворити рядок у масив символів.

```
String str1 = new String(new char[] {'h', 'e', 'l', 'l', 'o'});  
char[] helloArray = str1.toCharArray();
```

Введення в рядки. Клас String

Рядок може бути порожнім. Для цього йому можна присвоїти порожні лапки або видалити зі стоки всі символи.

```
String s = ""; // порожній рядок
if(s.length() == 0) System.out.println("String is empty");
```

У цьому випадку довжина рядка, що повертається методом **length()**, дорівнює 0. Клас **String** має спеціальний метод, який дозволяє перевірити рядок на порожнечу - **isEmpty()**. Якщо рядок порожній, він повертає true:

```
String s = ""; // порожній рядок
if(s.isEmpty()) System.out.println("String is empty");
```

Введення в рядки. Клас String

Змінна **String** може не вказувати на будь-який об'єкт і мати значення null:

```
String s = null;    // рядок не вказує на об'єкт
if(s == null) System.out.println("String is null");
```

Значення **null** не еквівалентне порожньому рядку. Наприклад, у такому випадку ми зіткнемося з помилкою виконання:

```
String s = null;    // рядок не вказує на об'єкт
if(s.length()==0) System.out.println("String is empty"); // !Помилка
```

Введення в рядки. Клас String

Оскільки змінна не вказує ні на який об'єкт **String**, то відповідно ми не можемо звертатися до методів об'єкта **String**. Щоб уникнути подібних помилок, можна попередньо перевіряти рядок на null:

```
String s = null; // рядок не вказує на об'єкт
if(s==null || s.length()==0) System.out.println("String is empty");
```

Основні методи класу String

Основні операції з рядками розкриваються через методи класу String, серед яких можна виділити такі:

- **concat()**: об'єднує рядки;
- **contains()**: перевіряє, чи містить рядок підрядок;
- **valueOf()**: перетворює об'єкт у рядковий вигляд;
- **join()**: з'єднує рядки з урахуванням роздільника;
- **compareTo()**: порівнює два рядки;

Введення в рядки. Клас String

- **endsWith()**: визначає, чи закінчується рядок на певний підрядок;
- **replace()**: замінює в рядку один підрядок на інший;
- **trim()**: видаляє початкові та кінцеві пробіли;
- **substring()**: повертає підрядок, починаючи з певного індексу до кінця або до певного індексу;
- **toLowerCase()**: переводить усі символи рядка в нижній регістр;
- **toUpperCase()**: переводить усі символи рядка у верхній регістр;
- **charAt()**: повертає символ рядка за індексом;
- **getChars()**: повертає групу символів;
- **equals()**: порівнює рядки з урахуванням регістру;
- **equalsIgnoreCase()**: порівнює рядки без урахування регістру;
- **regionMatches()**: порівнює підрядки в рядках;
- **indexOf()**: знаходить індекс першого входження підрядка в рядок;
- **lastIndexOf()**: знаходить індекс останнього входження підрядка в рядок;
- **startsWith()**: визначає, чи починається рядок із підрядка.

РОЗБЕРЕМО РОБОТУ ЦИХ МЕТОДІВ.

Основні операції з рядками

З'єднання рядків

Для з'єднання рядків можна використовувати операцію додавання ("+"):

```
String str1 = "Java";  
String str2 = "Hello";  
String str3 = str1 + " " + str2;  
  
System.out.println(str3); // Hello Java
```

При цьому якщо в операції додавання рядків використовується нерядковий об'єкт, наприклад, число, то цей об'єкт перетворюється на рядок:

```
String str3 = "Pik " + 2015;
```

Основні операції з рядками

Фактично ж під час додавання рядків з нерядковими об'єктами буде викликатися метод **valueOf()** класу **String**. Цей метод має безліч перевантажень і перетворює практично всі типи даних до рядка. Для перетворення об'єктів різних класів метод **valueOf** викликає метод **toString()** цих класів. Інший спосіб об'єднання рядків представляє метод **concat()**:

```
String str1 = "Java";  
String str2 = "Hello";  
str2 = str2.concat(str1); // HelloJava
```

Метод **concat()** приймає рядок, з яким треба об'єднати викликаючий рядок, і повертає з'єднаний рядок.

Основні операції з рядками

Ще один метод об'єднання - метод **join()** дозволяє об'єднати рядки з урахуванням роздільника. Наприклад, вище два рядки зливалися в одне слово "**HelloJava**", але в ідеалі ми б хотіли, щоб два підрядки були розділені пропуском. І для цього використовуємо метод **join()**:

```
String str1 = "Java";  
String str2 = "Hello";  
String str3 = String.join(" ", str2, str1); // Hello Java
```

Метод `join` є **статичним**. Першим параметром іде роздільник, яким розділятимуться підрядки в загальному рядку, а всі наступні параметри передають через кому довільний набір об'єднуваних підрядків - у цьому випадку два рядки, хоча їх може бути й більше.

Основні операції з рядками

Вилучення символів і підрядків

Для вилучення символів за індексом у класі `String` визначено метод **`char charAt(int index)`**. Він приймає індекс, за яким треба отримати символів, і повертає витягнутий символ:

```
String str = "Java";  
char c = str.charAt(2);  
System.out.println(c); // v
```

Як і в масивах, індексація починається з нуля.

Якщо треба витягти відразу групу символів або підрядок, то можна використовувати метод **`getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)`**. Він приймає такі параметри:

Основні операції з рядками

- **srcBegin**: індекс у рядку, з якого починається вилучення символів;
- **srcEnd**: індекс у рядку, до якого йде витяг символів;
- **dst**: масив символів, у який будуть витягуватися символи;
- **dstBegin**: індекс у масиві dst, з якого треба додавати витягнуті з рядка символи.

```
String str = "Hello world!";  
int start = 6;  
int end = 11;  
char[] dst= new char[end - start];  
str.getChars(start, end, dst, dstBegin: 0);  
System.out.println(dst); // world
```

Основні операції з рядками

Порівняння рядків

Для порівняння рядків використовуються методи **equals()** (з урахуванням регістру) і **equalsIgnoreCase()** (без урахування регістру). Обидва методи як параметр приймають рядок, з яким треба порівняти:

```
String str1 = "Hello";  
String str2 = "hello";  
  
System.out.println(str1.equals(str2)); // false  
System.out.println(str1.equalsIgnoreCase(str2)); // true
```

На відміну від порівняння числових та інших даних примітивних типів для рядків не застосовується знак рівності `==`. Замість нього треба використовувати метод **equals()**.

Основні операції з рядками

Ще один спеціальний метод **regionMatches()** порівнює окремі підрядки в межах двох рядків. Він має такі форми:

```
boolean regionMatches(int toffset, String other, int ooffset, int len)
boolean regionMatches(boolean ignoreCase, int toffset, String other, int
ooffset, int len)
```

Метод приймає такі параметри:

- **ignoreCase**: чи треба ігнорувати регістр символів під час порівняння. Якщо значення true, регістр ігнорується;
- **toffset**: початковий індекс у викликаючому рядку, з якого почнеться порівняння other: рядок, з яким порівнюється той, що викликає;
- **ooffset**: початковий індекс у порівнюваному рядку, з якого почнеться порівняння;
- **len**: кількість порівнюваних символів в обох рядках.

Основні операції з рядками

Використовуємо метод:

```
String str1 = "Hello world";  
String str2 = "I work";  
boolean result = str1.regionMatches( toffset: 6, str2,  ooffset: 2,  len: 3);  
System.out.println(result); // true
```

У цьому випадку метод порівнює 3 символи з 6-го індексу першого рядка ("wor") і 3 символи з 2-го індексу другого рядка ("wor"). Оскільки ці підрядки однакові, то повертається true.

І ще одна пара методів **int compareTo(String str)** і **int compareToIgnoreCase(String str)** також дають змогу порівняти два рядки, але водночас вони також дають змогу дізнатися, чи більший один рядок, ніж інший, чи ні. Якщо значення, що повертається, більше за 0, то перший рядок більший за другий, якщо менше нуля, то, навпаки, другий більший за перший. Якщо рядки рівні, то повертається 0.

Основні операції з рядками

Для визначення більший чи менший один рядок, ніж інший, використовується лексикографічний порядок. Тобто, наприклад, рядок "A" менший, ніж рядок "B", оскільки символ 'A' в алфавіті стоїть перед символом 'B'. Якщо перші символи рядків рівні, то в розрахунок беруться наступні символи. Наприклад:

```
String str1 = "hello";  
String str2 = "world";  
String str3 = "hell";  
  
System.out.println(str1.compareTo(str2)); // -15 - str1 менше ніж str2  
System.out.println(str1.compareTo(str3)); // 1 - str1 більше ніж str3
```

Пошук у рядку

Метод **indexOf()** знаходить індекс першого входження підрядка в рядок, а метод **lastIndexOf()** - індекс останнього входження. Якщо підрядок не буде знайдено, то обидва методи повертають -1:

Основні операції з рядками

```
String str = "Hello world";  
int index1 = str.indexOf('l'); // 2  
int index2 = str.indexOf("wo"); //6  
int index3 = str.lastIndexOf(ch: 'l'); //9
```

Метод **startsWith()** дає змогу визначити, чи починається рядок з певного підрядка, а метод **endsWith()** дає змогу визначити, чи закінчується рядок на певний підрядок:

```
String str = "myfile.exe";  
boolean start = str.startsWith("my"); //true  
boolean end = str.endsWith("exe"); //true
```

Заміна в рядку

Метод **replace()** дозволяє замінити в рядку одну послідовність символів на іншу:

Основні операції з рядками

```
String str = "Hello world";  
String replStr1 = str.replace( oldChar: 'l', newChar: 'd'); // Heddo wordd  
String replStr2 = str.replace( target: "Hello", replacement: "Bye"); // Bye world
```

Обрізка рядка

Метод `trim()` дає змогу видалити початкові та кінцеві пробіли:

```
String str = "  hello world  ";  
str = str.trim(); // hello world
```

Метод **`substring()`** повертає підрядок, починаючи з певного індексу до кінця або до певного індексу:

```
String str = "Hello world";  
String substr1 = str.substring( beginIndex: 6); // world  
String substr2 = str.substring(3,5); //lo
```

Основні операції з рядками

Зміна регістру

Метод `toLowerCase()` переводить усі символи рядка в нижній регістр, а метод `toUpperCase()` - у верхній:

```
String str = "Hello World";  
System.out.println(str.toLowerCase()); // hello world  
System.out.println(str.toUpperCase()); // HELLO WORLD
```

Split

Метод `split()` дозволяє розбити рядок на підрядки за певним роздільником. Роздільник - який-небудь символ або набір символів передається як параметр у метод. Наприклад, розіб'ємо текст на окремі слова:

```
String text = "FIFA will never regret it";  
String[] words = text.split(regex: " ");  
for(String word : words)  
    System.out.println(word);
```

У цьому випадку рядок буде розділятися по пробілу

StringBuffer та StringBuilder

Об'єкти String є **незмінними**, тому всі операції, які змінюють рядки, фактично призводять до створення нового рядка, що позначається на продуктивності програми. Для розв'язання цієї проблеми, щоб робота з рядками відбувалася з меншими витратами, в Java було додано класи **StringBuffer** і **StringBuilder**. По суті вони нагадують розширюваний рядок, який можна змінювати без шкоди для продуктивності.

Ці класи схожі, практично двійники, вони мають однакові конструктори, одні й ті самі методи, які однаково використовуються. Єдина їхня відмінність полягає в тому, що клас **StringBuffer** синхронізований і потокобезпечний. Тобто клас **StringBuffer** зручніше використовувати в багатопотокових додатках, де об'єкт цього класу може змінюватися в різних потоках. Якщо ж мова про багатопотокові додатки не йде, то краще використовувати клас **StringBuilder**, який не є потокобезпечним, але при цьому працює швидше, ніж **StringBuffer** в однопотокових додатках. **StringBuffer визначає чотири конструктори:**

StringBuffer та StringBuilder

- `StringBuffer()`
- `StringBuffer(int capacity)`
- `StringBuffer(String str)`
- `StringBuffer(CharSequence chars)`

Аналогічні конструктори визначає `StringBuilder`:

- `StringBuilder()`
- `StringBuilder(int capacity)`
- `StringBuilder(String str)`
- `StringBuilder(CharSequence chars)`

Розглянемо роботу цих класів на прикладі функціональності **StringBuffer**. При всіх операціях з рядками **StringBuffer** / **StringBuilder** перерозподіляє виділену пам'ять. І щоб уникнути занадто частого перерозподілу пам'яті, **StringBuffer** / **StringBuilder** заздалегідь резервує деяку ділянку пам'яті, яка може використовуватися. Конструктор без параметрів резервує в пам'яті місце для 16 символів.

StringBuffer та StringBuilder

Якщо ми хочемо, щоб кількість символів була іншою, то ми можемо застосувати другий конструктор, який як параметр приймає кількість символів.

Третій і четвертий конструктори обох класів приймають рядок і набір символів, при цьому резервуючи пам'ять для додаткових 16 символів.

За допомогою методу `capacity()` ми можемо отримати кількість символів, для яких зарезеровано пам'ять. А за допомогою методу **`ensureCapacity()`** змінити мінімальну ємність буфера символів:

```
String str = "Java";
StringBuffer strBuffer = new StringBuffer(str);
System.out.println("Ємність: " + strBuffer.capacity()); // 20
strBuffer.ensureCapacity(minimumCapacity: 32);
System.out.println("Ємність: " + strBuffer.capacity()); // 42
System.out.println("Длина: " + strBuffer.length()); // 4
```

StringBuffer та StringBuilder

Оскільки на самому початку StringBuffer ініціалізується рядком "Java", то його ємність становить $4 + 16 = 20$ символів. Потім ми збільшуємо ємність буфера за допомогою виклику **strBuffer.ensureCapacity(32)** підвищуємо мінімальну ємність буфера до 32 символів. Однак фінальна ємність може відрізнятись в більшу сторону. Так, у цьому випадку я отримую ємність не 32 і не $32 + 4 = 36$, а 42 символи. Річ у тім, що з метою підвищення ефективності Java може додатково виділяти пам'ять.

Але в будь-якому разі незалежно від ємності довжина рядка, яку можна отримати за допомогою методу `length()`, у StringBuffer залишається незмінною - 4 символи (оскільки в "Java" 4 символи).

Щоб отримати рядок, який зберігається в StringBuffer, ми можемо використовувати стандартний метод `toString()`:

```
String str = "Java";  
StringBuffer strBuffer = new StringBuffer(str);  
System.out.println(strBuffer.toString()); // Java
```

За всіма своїми операціями StringBuffer і StringBuilder нагадують клас String.

StringBuffer та StringBuilder

Отримання та встановлення символів

Метод `charAt()` отримує, а метод `setCharAt()` встановлює символ за певним індексом:

```
StringBuffer strBuffer = new StringBuffer("Java");  
char c = strBuffer.charAt(0); // J  
System.out.println(c);  
strBuffer.setCharAt(index: 0, ch: 'c');  
System.out.println(strBuffer.toString()); // cava
```

Метод **`getChars()`** отримує набір символів між визначеними індексами:

```
StringBuffer strBuffer = new StringBuffer("world");  
int startIndex = 1;  
int endIndex = 4;  
char[] buffer = new char[endIndex - startIndex];  
strBuffer.getChars(startIndex, endIndex, buffer, dstBegin: 0);  
System.out.println(buffer); // orl
```

StringBuffer та StringBuilder

Додавання в рядок

Метод `append()` додає підрядок у кінець `StringBuffer`:

```
StringBuffer strBuffer = new StringBuffer("hello");  
strBuffer.append(" world");  
System.out.println(strBuffer.toString()); // hello world
```

Метод **`insert()`** додає рядок або символ за певним індексом у `StringBuffer`:

```
StringBuffer strBuffer = new StringBuffer("word");  
  
strBuffer.insert( offset: 3, c: 'l');  
System.out.println(strBuffer.toString()); //world  
  
strBuffer.insert( offset: 0, str: "s");  
System.out.println(strBuffer.toString()); //sworld
```


StringBuffer та StringBuilder

Видалення символів

Метод **delete()** видаляє всі символи з певного індексу про певну позицію, а метод **deleteCharAt()** видаляє один символ за певним індексом:

```
StringBuffer strBuffer = new StringBuffer("assembler");  
strBuffer.delete(0,2);  
System.out.println(strBuffer.toString()); //sembler  
  
strBuffer.deleteCharAt(index: 6);  
System.out.println(strBuffer.toString()); //semble
```

Обрізання рядка

Метод **substring()** обрізає рядок з певного індексу до кінця, або до певного індексу:

StringBuffer та StringBuilder

```
StringBuffer strBuffer = new StringBuffer("hello java!");  
String str1 = strBuffer.substring( start: 6); // обрізка рядка з 6 символу до кінця  
System.out.println(str1); //java!  
  
String str2 = strBuffer.substring(3, 9); // обрізка рядка з 3 по 9 символ  
System.out.println(str2); //lo jav
```

Зміна довжини

Для зміни довжини StringBuffer (не ємності буфера символів) застосовується метод `setLength()`. Якщо StringBuffer збільшується, то його рядок просто доповнюється в кінці порожніми символами, якщо зменшується - то рядок по суті обрізається:

```
StringBuffer strBuffer = new StringBuffer("hello");  
strBuffer.setLength(10);  
System.out.println(strBuffer.toString()); //"hello      "  
  
strBuffer.setLength(4);  
System.out.println(strBuffer.toString()); //"hell"
```

StringBuffer та StringBuilder

Заміна в рядку

Для заміни підрядка між певними позиціями в **StringBuffer** на інший підрядок застосовується метод **replace()**:

```
StringBuffer strBuffer = new StringBuffer("hello world!");  
strBuffer.replace( start: 6, end: 11, str: "java");  
System.out.println(strBuffer.toString()); //hello java!
```

Перший параметр методу `replace` вказує, з якої позиції треба почати заміну, другий параметр - до якої позиції, а третій параметр вказує на підрядок заміни.

Зворотний порядок у рядку

Метод **reverse()** змінює порядок у **StringBuffer** на зворотний:

```
StringBuffer strBuffer = new StringBuffer("assembler");  
strBuffer.reverse();  
System.out.println(strBuffer.toString()); //relbmess
```

Регулярні вирази

Регулярні вирази (regular expressions - regex) являють собою потужний інструмент для обробки рядків. Регулярні вирази дають змогу задати шаблон, якому має відповідати рядок або підрядок.

Деякі методи класу `String` приймають регулярні вирази і використовують їх для виконання операцій над рядками.

split

Для поділу рядка на підрядки застосовується метод `split()`. Як параметр він може приймати регулярний вираз, який представляє критерій поділу рядка. Наприклад, розділимо речення на слова:

```
String text = "FIFA will never regret it";
String[] words = text.split(regex: "\\s*(\\s|,|!|\\.|)\\s*");
for(String word : words){
    System.out.println(word);
}
```

Регулярні вирази

Для поділу застосовується регулярний вираз `"\\s*(\\s|,|!|\\\\.)*\\s"`. Підвираз `"\\s"` по суті представляє пробіл. Зірочка вказує, що символ може бути присутнім від 0 до нескінченної кількості разів. Тобто додаємо зірочку і ми отримуємо невизначену кількість пропусків, які йдуть підряд, - `"\\s*"` (тобто неважливо, скільки пропусків між словами). Причому пробілів може взагалі не бути. У дужках вказує група виразів, яка може йти після невизначеної кількості пробілів.

Група дозволяє нам визначити набір значень через вертикальну риску, і підрядок має відповідати одному з цих значень. Тобто в групі `"\\s|,|!|\\\\."` підрядок може відповідати пробілу, комі, знаку оклику або крапці. Причому оскільки крапка являє собою спеціальну послідовність, то, щоб вказати, що ми маємо на увазі саме знак крапки, а не спеціальну послідовність, перед крапкою ставимо слеші.

Регулярні вирази

Відповідність рядка. Matches

Ще один метод класу String - **matches()** приймає регулярний вираз і повертає true, якщо рядок відповідає цьому виразу. Інакше повертає false. Наприклад, перевіримо, чи відповідає рядок номеру телефону:

```
String input = "+12343454556";
boolean result = input.matches(regex: "(\\+*)\\d{11}");
if(result)
    System.out.println("It is a phone number");
else
    System.out.println("It is not a phone number!");
```

У цьому випадку в регулярному виразі спочатку визначається група "(\\+*)". Тобто спочатку може йти знак плюса, але також він може бути відсутнім. Далі дивимося, чи відповідають наступні 11 символів цифрам. Вираз "\\d" представляє цифровий символ, а число у фігурних дужках - {11} - скільки разів цей тип символів має повторюватися. Тобто ми шукаємо рядок, де спочатку може йти знак плюс (або він може бути відсутнім), а потім йде 11 цифрових символів.

Регулярні вирази

Клас Pattern

Велика частина функціональності по роботі з регулярними виразами в Java зосереджена в пакеті **java.util.regex**. Сам регулярний вираз являє собою шаблон для пошуку збігів у рядку. Для задавання подібного шаблону і пошуку підрядків у рядку, які задовольняють даному шаблону, в Java визначено класи Pattern і Matcher. Для простого пошуку відповідностей у класі Pattern визначено статичний метод **boolean matches(String pattern, CharSequence input)**. Цей метод повертає true, якщо послідовність символів input повністю відповідає шаблону рядка pattern:

```
String input = "Hello";
boolean found = Pattern.matches("Hello", input);
if(found)
    System.out.println("Знайдено");
else
    System.out.println("Не знайдено");
```

Але, як правило, для пошуку відповідностей застосовується інший спосіб - використання класу Matcher.

Регулярні вирази

Клас **Matcher**

Основні методи класу **Matcher**:

- **boolean matches()**: повертає true, якщо весь рядок збігається з шаблоном;
- **boolean find()**: повертає true, якщо в рядку є підрядок, який збігається з шаблоном, і переходить до цього підрядка;
- **String group()**: повертає підрядок, який збігся з шаблоном у результаті виклику методу find. Якщо збіг відсутній, то метод генерує виняток `IllegalStateException`;
- **int start()**: повертає індекс поточного збігу;
- **int end()**: повертає індекс наступного збігу після поточного;
- **String replaceAll(String str)**: замінює всі знайдені збіги підрядком str і повертає змінений рядок з урахуванням замін.

Регулярні вирази

Використовуємо клас **Matcher**. Для цього спочатку треба створити об'єкт **Pattern** за допомогою статичного методу **compile()**, який дозволяє встановити шаблон:

```
Pattern pattern = Pattern.compile( regex: "Hello");
```

Як шаблон виступає рядок "Hello". Метод **compile()** повертає об'єкт **Pattern**, який ми потім можемо використовувати в програмі. У класі **Pattern** також визначено метод **matcher(String input)**, який як параметр приймає рядок, де треба проводити пошук, і повертає об'єкт **Matcher**:

```
String input = "Hello world! Hello Java!";  
Pattern pattern = Pattern.compile( regex: "hello");  
Matcher matcher = pattern.matcher(input);
```

Регулярні вирази

Потім у об'єкта `Matcher` викликається метод `matches()` для пошуку відповідностей шаблону в тексті:

```
String input = "Hello";
Pattern pattern = Pattern.compile(regex: "Hello");
Matcher matcher = pattern.matcher(input);
boolean found = matcher.matches();
if(found)
    System.out.println("Знайдено");
else
    System.out.println("Не знайдено");
```

Розглянемо більш функціональний приклад зі знаходженням не повної відповідності, а окремих збігів у рядку:

```
String input = "Hello Java! Hello JavaScript! JavaSE 8.";
Pattern pattern = Pattern.compile(regex: "Java(\\w*)");
Matcher matcher = pattern.matcher(input);
while(matcher.find())
    System.out.println(matcher.group());
```

Регулярні вирази

Припустимо, ми хочемо знайти в рядку всі входження слова Java. У вихідному рядку це три слова: "Java", "JavaScript" і "JavaSE". Для цього застосуємо шаблон "Java(\\w*)". Цей шаблон використовує синтаксис регулярних виразів. Слово "Java" на початку говорить про те, що всі збіги в рядку повинні починатися на Java. Вираз (\\w*) означає, що після "Java" у збігу може бути будь-яка кількість алфавітно-цифрових символів. Вираз \\w означає алфавітно-цифровий символ, а зірочка після виразу вказує на невизначену їхню кількість - їх може бути один, два, три або взагалі не бути. І щоб java не розглядала \\w як ескейп-послідовність, як \\n, то вираз екранується ще одним слешем.

Далі застосовується метод **find()** класу `Matcher`, який дає змогу переходити до наступного збігу в рядку. Тобто перший виклик цього методу знайде перший збіг у рядку, другий виклик знайде другий збіг тощо. Тобто за допомогою циклу **while(matcher.find())** ми можемо пройтися по всіх збігах. Кожен збіг ми можемо отримати за допомогою методу **matcher.group()**. У підсумку програма видасть такий результат:

Регулярні вирази

Заміна в рядку

Тепер зробимо заміну всіх збігів за допомогою методу `replaceAll()`:

```
String input = "Hello Java! Hello JavaScript! JavaSE 8.";
Pattern pattern = Pattern.compile( regex: "Java(\\w*)" );
Matcher matcher = pattern.matcher(input);
String newStr = matcher.replaceAll( replacement: "HTML" );
System.out.println(newStr); // Hello HTML! Hello HTML! HTML 8.
```

Також треба зазначити, що в класі `String` також є метод `replaceAll()` з подібною дією:

```
String input = "Hello Java! Hello JavaScript! JavaSE 8.";
String myStr = input.replaceAll( regex: "Java(\\w*)", replacement: "HTML" );
System.out.println(myStr); // Hello HTML! Hello HTML! HTML 8.
```

Регулярні вирази

Загальні відповідні символи

Regular Expression	Description
.	Matches any character
^regex	Finds regex that must match at the beginning of the line.
regex\$	Finds regex that must match at the end of the line.
[abc]	Set definition, can match the letter a or b or c.
[abc] [vz]	Set definition, can match a or b or c followed by either v or z.
[^abc]	When a caret appears as the first character inside square brackets, it negates the pattern. This pattern matches any character except a or b or c.
[a-d1-7]	Ranges: matches a letter between a and d and figures from 1 to 7, but not d1.
x z	Finds X or Z.
xz	Finds X directly followed by Z.
\$	Checks if a line end follows.

Регулярні вирази

Мета-символи

Regular Expression	Description
<code>\d</code>	Any digit, short for <code>[0-9]</code>
<code>\D</code>	A non-digit, short for <code>[^0-9]</code>
<code>\s</code>	A whitespace character, short for <code>[\t\n\x0b\r\f]</code>
<code>\S</code>	A non-whitespace character, short for
<code>\w</code>	A word character, short for <code>[a-zA-Z_0-9]</code>
<code>\W</code>	A non-word character <code>[^\w]</code>
<code>\S+</code>	Several non-whitespace characters
<code>\b</code>	Matches a word boundary where a word character is <code>[a-zA-Z0-9_]</code>

Контрольні питання

- Зазначте визначення терміну "рядок" в мові програмування Java.
- Виокреміть різницю між класами String та StringBuffer у мові програмування Java.
- Визначте відмінності між класами StringBuffer та StringBuilder в контексті мови програмування Java.
- Як у мові програмування Java можна проводити порівняння двох рядків? Розкрийте різницю між виразами `str1 == str2` та `str1.equals(str2)` в Java.
- Як здійснити реверс рядка в мові програмування Java?
- Чи можна вважати рядки незмінними або кінцевими в Java? Якщо так, то які переваги надає незмінність рядків?
- Визначте поняття "регулярні вирази" в контексті мови програмування Java.

The END
Lec 7.

.NET та JAVA #8

Обробка виключень

1. Введення в обробку виключень
2. Оператор throws
3. Автоматичне управління ресурсами
4. Класи виключень
5. Як JVM обробляє виключення
6. Створення користувальницьких виключень

Введення в обробку виключень

Нерідко в процесі виконання програми можуть виникати помилки, при тому необов'язково з вини розробника. Деякі з них важко передбачити або передбачити, а іноді й зовсім неможливо. Так, наприклад, може несподівано обірватися мережеве з'єднання під час передачі файлу. Подібні ситуації називаються **винятками (Exceptions)**.

Обробка виключень в Java є одним з потужних механізмів для обробки помилок під час виконання, що дозволяє підтримувати нормальну роботу програми.

Передбачено спеціальні засоби для обробки подібних ситуацій. Одним із таких засобів є конструкція **try...catch...finally**. У разі виникнення винятку в блоці try керування переходить у блок catch, який може обробити цей виняток. Якщо такого блоку не знайдено, то користувачеві відображається повідомлення про необроблене виключення, а подальше виконання програми зупиняється. І щоб подібної зупинки не сталося, і треба використовувати блок try..catch.

Введення в обробку виключень

Ключові слова винятків Java

Ключове слово	Опис
try	Ключове слово try використовується для вказівки блоку, в якому ми повинні розмістити код виключення. Це означає, що ми не можемо використовувати тільки блок try. За блоком try повинен слідувати або catch, або finally.
catch	Блок catch використовується для обробки виключення. Йому повинен передувати блок try, що означає, що ми не можемо використовувати блок catch окремо. За ним може слідувати блок finally.
finally	Блок finally використовується для виконання необхідного коду програми. Він виконується незалежно від того, обробляється виключення чи ні.
throw	Ключове слово throw використовується для генерування виключення.
throws	Ключове слово throws використовується для оголошення винятків. Воно вказує на те, що в методі може виникнути виключення. Воно не генерує виключення. Воно завжди використовується з сигнатурою методу.

Введення в обробку виключень

```
int[] numbers = new int[3];  
numbers[4]=45;  
System.out.println(numbers[4]);
```

Оскільки в нас масив numbers може містити тільки 3 елементи, то під час виконання інструкції numbers[4]=45 консоль відобразить виняток, і виконання програми буде завершено.

```
try{  
    int[] numbers = new int[3];  
    numbers[4]=45;  
    System.out.println(numbers[4]);  
}  
catch(Exception ex){  
    ex.printStackTrace();  
}  
System.out.println("Complete");
```

Під час використання блоку try...catch спочатку виконуються всі інструкції між операторами try і catch. Якщо в блоці try раптом виникає виняток, то звичайний порядок виконання зупиняється і переходить до інструкції catch. Тому коли виконання програми дійде до рядка numbers[4]=45 - програма зупиниться і перейде до блоку catch.

Введення в обробку виключень

Вираз `catch` має такий синтаксис:

catch (тип_виключення ім'я_змінної).

У цьому випадку оголошується змінна `ex`, яка має тип **Exception**. Але якщо виняток, що виник, не є винятком типу, зазначеного в інструкції **catch**, то він не обробляється, а програма просто зависає або викидає повідомлення про помилку.

Але оскільки тип **Exception** є базовим класом для всіх винятків, то вираз `catch (Exception ex)` буде обробляти практично всі винятки. Обробка ж винятку в цьому випадку зводиться до виведення на консоль стека трасування помилки за допомогою методу **printStackTrace()**, визначеного в класі **Exception**.

Введення в обробку виключень

Після завершення виконання блоку **catch** програма продовжує свою роботу, виконуючи всі інші інструкції після блоку **catch**. Конструкція **try..catch** також може мати блок **finally**.

Однак цей блок **необов'язковий**, і його можна під час обробки винятків опускати. Блок **finally** виконується в будь-якому разі, виникло виключення в блоці try чи ні.

```
try{
    int[] numbers = new int[3];
    numbers[4]=45;
    System.out.println(numbers[4]);
}
catch(Exception ex){
    ex.printStackTrace();
}
finally{
    System.out.println("Finally block");
}
System.out.println("Complete");
```

Введення в обробку виключень

Обробка кількох винятків

У Java є безліч різних типів винятків, і ми можемо розмежувати їх обробку, включивши додаткові блоки **catch**:

```
int[] numbers = new int[3];
try{
    numbers[6]=45;
    numbers[6]=Integer.parseInt(s: "gfd");
}
catch(ArrayIndexOutOfBoundsException ex){
    System.out.println("Вихід за межі масиву");
}
catch(NumberFormatException ex){
    System.out.println("Помилка перетворення з рядка в число");
}
```

Якщо у нас виникає виняток певного типу, то він переходить до відповідного блоку catch.

Введення в обробку виключень

Оператор throw

Щоб повідомити про виконання виняткових ситуацій у програмі, можна використовувати оператор **throw**. Тобто за допомогою цього оператора ми самі можемо створити виняток і викликати його в процесі виконання. Наприклад, у нашій програмі відбувається введення числа, і ми хочемо, щоб, якщо число більше 30, то виникало виключення:

```
try{
    Scanner in = new Scanner(System.in);
    int x = in.nextInt();
    if(x>=30){
        throw new Exception("Число x має бути меншим за 30");
    }
}
catch(Exception ex){
    System.out.println(ex.getMessage());
}
System.out.println("Complete");
```

Тут для створення об'єкта винятку використовується конструктор класу **Exception**, у який передається повідомлення про виняток. І якщо число x виявиться більшим за 29, то буде викинуто виняток і управління перейде до блоку **catch**. У блоці catch ми можемо отримати повідомлення про виключення за допомогою методу **getMessage()**.

Оператор throws

Іноді метод, у якому може генеруватися виняток, сам не обробляє цей виняток. У цьому випадку в оголошенні методу використовується оператор **throws**, який треба обробити під час виклику цього методу. Наприклад, у нас є метод обчислення факторіалу, і нам треба обробити ситуацію, якщо в метод передається число, менше ніж 1:

```
public static int getFactorial(int num) throws Exception{
    if(num<1) throw new Exception("The number is less than 1");
    int result=1;
    for(int i=1; i<=num;i++) result*=i;

    return result;
}
```

За допомогою оператора **throw** за умовою викидається виняток. Водночас метод сам це виняток не обробляє за допомогою **try..catch**, тому у визначенні методу використовується вираз **throws Exception**. Тепер при виклику цього методу нам обов'язково треба обробити викинуте виключення:

Оператор throws

```
try{
    int result = getFactorial( num: -6);
    System.out.println(result);
}
catch(Exception ex){
    System.out.println(ex.getMessage());
}
```

Без обробки винятку у нас виникне помилка компіляції, і ми не зможемо скомпілювати програму. Як альтернативу ми могли б і не використовувати оператор **throws**, а обробити виняток прямо в методі:

```
public static int getFactorial(int num){
    int result = 1;
    try{
        if(num<1) throw new Exception("The number is less than 1");
        for(int i=1; i<=num;i++){
            result*=i;
        }
    }
    catch(Exception ex){
        System.out.println(ex.getMessage());
        result=num;
    }
    return result;
}
```

Автоматичне управління ресурсами

Якщо ви перехоплюєте за допомогою **catch** багато винятків в одному блоці **try**, ви помітите, що код блоку перехоплення здебільшого складається з надлишкового коду для реєстрації помилки.

У **Java 7** однією з можливостей став покращений блок **catch**, в якому ми можемо перехоплювати декілька винятків в одному блоці **catch**. Ось приклад блоку **catch** з такою можливістю:

```
catch (IOException | SQLException ex) {  
    throw new MyException(ex.getMessage());  
}
```

Існують деякі обмеження, наприклад, об'єкт виключення є фінальним, і ми не можемо модифікувати його всередині блоку перехоплення, читайте повний аналіз у статті (посилання нижче).

Автоматичне управління ресурсами

Здебільшого ми використовуємо блок **finally** лише для того, щоб закрити ресурси. Іноді ми забуваємо їх закрити і отримуємо виключення під час виконання, коли ресурси вичерпуються. Ці винятки важко налагоджувати, і нам може знадобитися перевірити кожне місце, де ми використовуємо ресурс, щоб переконатися, що ми закриваємо його.

У **Java 7** одним з покращень стала спроба з ресурсами, де ми можемо створити ресурс в самому операторі `try` і використовувати його в блоці **try-catch**. Коли виконання виходить з блоку **try-catch**, середовище виконання автоматично закриває ці ресурси.

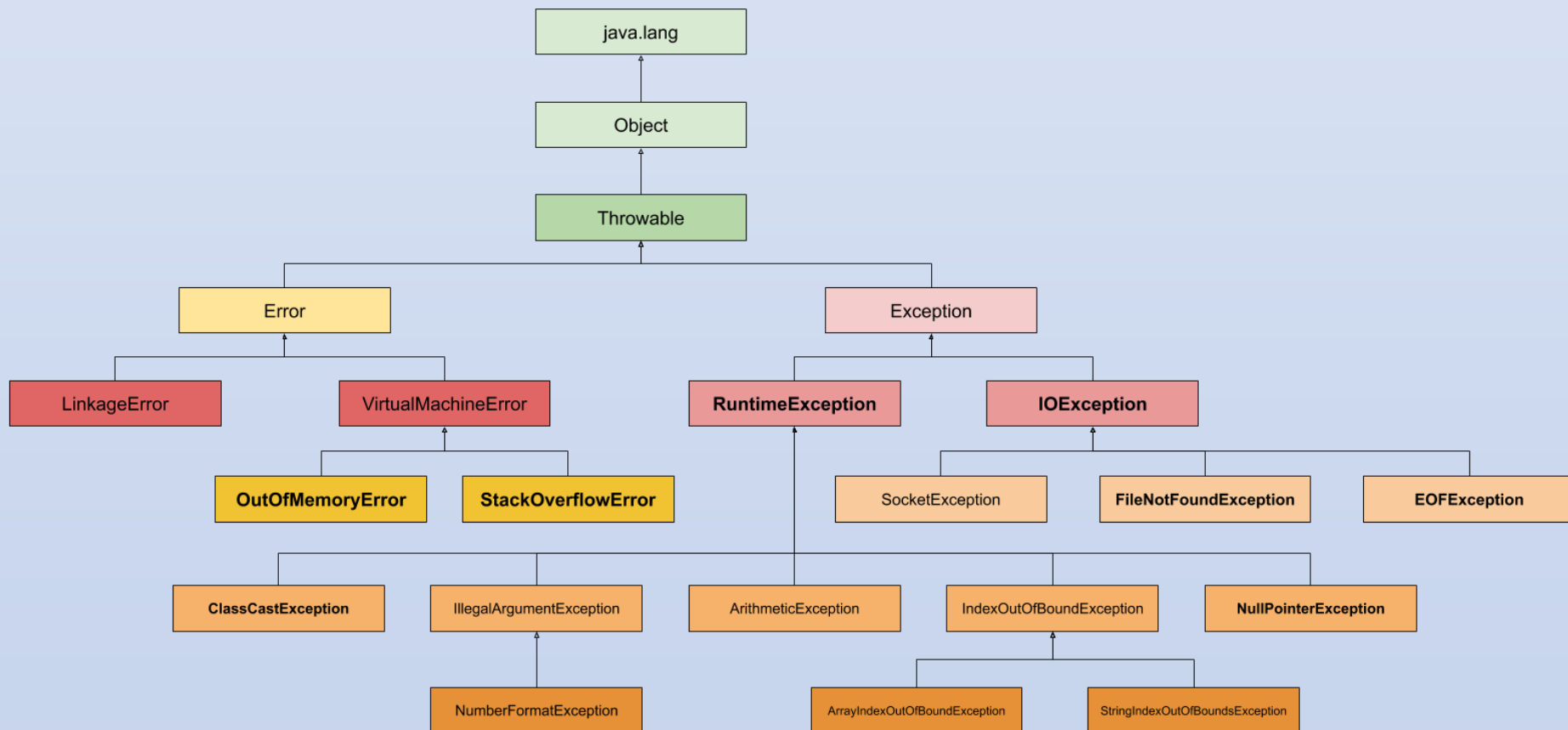
Ця конструкція має назву **try-with-resources**

Її ми розглянемо в лекції про роботу з файловою системою!

Класи виключень

Базовим класом для всіх винятків є клас **Throwable**. Від нього вже успадковуються два класи: **Error** і **Exception**.

Усі інші класи є похідними від цих двох класів.



Класи виключень

Error - це виняткові сценарії, які виходять за рамки програми, і які неможливо передбачити та відновити після них. Наприклад, апаратний збій, збій віртуальної машини **Java (JVM)** або помилка, пов'язана з нестачею пам'яті. Ось чому у нас є окрема ієрархія помилок, і ми не повинні намагатися впоратися з цими ситуаціями. Деякі з найпоширеніших помилок - **OutOfMemoryError** та **StackOverflowError**.

Checked Exceptions - це виняткові ситуації, які ми можемо передбачити у програмі і спробувати відновитися після них. Наприклад, виключення **FileNotFoundException**. Ми повинні перехопити цей виняток і надати корисне повідомлення користувачеві, а також належним чином зареєструвати його для цілей налагодження. Виключення є батьківським класом для всіх винятків, що перевіряються. Якщо ми генеруємо контрольоване виключення, ми повинні перехопити його в тому ж методі, або передати його викликаючому користувачеві за допомогою ключового слова **throws**.

Runtime Exception - виникають через погане програмування. Наприклад, спроба отримати елемент з масиву. Ми повинні спочатку перевірити довжину масиву, перш ніж намагатися отримати елемент, інакше це може згенерувати виключення **ArrayIndexOutOfBoundsException** під час виконання. **RuntimeException** є батьківським класом для всіх винятків часу виконання. Якщо ми генеруємо будь-яке виключення часу виконання в методі, не обов'язково вказувати його в реченні **throws** сигнатури методу. Винятків часу виконання можна уникнути за допомогою кращого програмування.

Класи виключень

Exception та всі його підкласи не надають жодних специфічних методів, а всі методи визначені в базовому класі - **Throwable**. Класи винятків створені для визначення різних типів винятків, щоб ми могли легко визначити першопричину і обробити виняток відповідно до його типу. Клас **Throwable** реалізує інтерфейс **Serializable** для забезпечення сумісності.

Деякі з корисних методів класу **Throwable**:

- 1) **public String getMessage()** - цей метод повертає рядок-повідомлення класу **Throwable**, і це повідомлення може бути надане під час створення виключення через його конструктор.
- 2) **public String getLocalizedMessage()** - цей метод передбачено для того, щоб підкласи могли перевизначати його для надання специфічного для локалі повідомлення програмі, що викликає його. Реалізація цього методу у класі **Throwable** використовує метод **getMessage()** для повернення повідомлення про виключення.
- 3) **public synchronized Throwable getCause()** - цей метод повертає причину виключення або нуль, якщо причина невідома.
- 4) **public String toString()** - метод повертає інформацію про **Throwable** у форматі **String**, повернутий **String** містить ім'я класу **Throwable** та локалізоване повідомлення.
- 5) **public void printStackTrace()** - цей метод виводить інформацію про трасування стеку у стандартний потік помилок, цей метод перевантажений, і ми можемо передати **PrintStream** або **PrintWriter** як аргумент для запису інформації про трасування стеку у файл або потік.

Як JVM обробляє виключення

Обробка винятків за замовчуванням. Кожного разу, коли всередині методу виникає виключення, метод створює об'єкт, відомий як об'єкт виключення, і передає його системі виконання (**JVM**). Об'єкт виключення містить ім'я та опис виключення, а також поточний стан програми, в якій виникло виключення. Створення об'єкта виключення та обробка його у системі виконання називається генеруванням виключення. Може існувати список методів, які були викликані для того, щоб дістатися до методу, в якому виникло виключення. Цей впорядкований список методів називається стеком викликів. Тепер відбудеться наступна процедура:

- 1) Система виконання шукає у стеку викликів метод, який містить блок коду, що може обробити виключення, яке виникло. Цей блок коду називається обробником виключення.
- 2) Система виконання починає пошук з методу, в якому виникла виключна ситуація, і проходить по стеку викликів у зворотному порядку, в якому викликалися методи.
- 3) Якщо вона знаходить відповідний обробник, то передає йому виключення, що виникло. Відповідний обробник означає, що тип згенерованого об'єкта виключення відповідає типу об'єкта виключення, який він може обробити.
- 4) Якщо система виконання перебирає всі методи у стеку викликів і не знаходить відповідного обробника, то система виконання передає об'єкт виключення обробнику винятків за замовчуванням, який є частиною системи виконання. Цей обробник виводить інформацію про виключення у наступному форматі та аварійно завершує програму.

Створення користувальницьких класів

Оскільки всі класи винятків успадковуються від класу `Exception`, то всі вони успадковують низку його методів, які дають змогу отримати інформацію про характер винятку. Серед цих методів відзначимо найважливіші: Метод **`getMessage()`** повертає повідомлення про виключення

Метод **`getStackTrace()`** повертає масив, що містить трасування стека виключення Метод **`printStackTrace()`** відображає трасування стека. Наприклад:

```
try {  
    int x = 6/0;  
} catch (Exception ex){  
    ex.printStackTrace();  
}  
  
System.out.println("Program is here!");
```

```
java.lang.ArithmeticException Create breakpoint : / by zero  
    at ua.aharoo.Freshman.main(Test4.java:154)  
Program is here!
```

Створення користувальницьких класів

Хоча наявні в стандартній бібліотеці класів Java класи винятків описують більшість виняткових ситуацій, що можуть виникнути під час виконання програми, все ж таки іноді потрібно створити свої власні класи винятків зі своєю логікою. Щоб створити свій клас винятків, треба успадкувати його від класу **Exception**.

Наприклад, у нас є клас, що обчислює факторіал, і нам треба викидати спеціальний виняток, якщо число, що передається в метод, менше 1:

```
class FactorialException extends Exception{
    2 usages
    private int number;
    no usages
    public int getNumber(){return number;}
    1 usage
    public FactorialException(String message, int num){
        super(message);
        number=num;
    }
}
```

```
class Factorial{
    no usages
    public static int getFactorial(int num) throws FactorialException{
        int result=1;
        if(num<1) throw new FactorialException("The number is less than 1", num);
        for(int i=1; i<=num;i++) result*=i;

        return result;
    }
}
```

Створення користувальницьких класів

Тут для визначення помилки, пов'язаної з обчисленням факторіала, визначено клас **FactorialException**, який успадковується від **Exception** і який містить усю інформацію про обчислення. У конструкторі **FactorialException** у конструктор базового класу **Exception** передається повідомлення про помилку: `super(message)`. Крім того, окреме поле призначене для зберігання числа, факторіал якого обчислюється. Для генерації винятку в методі обчислення факторіалу викидається виняток за допомогою оператора **throw**: **throw new FactorialException("Число не може бути меншим за 1", num)**.

Крім того, оскільки це виняток не обробляється за допомогою **try...catch**, то ми передаємо обробку методу, що викликає, використовуючи оператор **throws**: **public static int getFactorial(int num) throws FactorialException**.

```
try{
    int result = Factorial.getFactorial( num: 6);
    System.out.println(result);
}
catch(FactorialException ex){
    System.out.println(ex.getMessage());
    System.out.println(ex.getNumber());
}
```

Контрольні питання

1. Надайте формальне визначення поняття "виключення" (Exception) в контексті програмування.
2. Розгляньте роботу блоків `try`, `catch`, `finally` в структурі обробки виключень в мові програмування.
3. Для чого конкретно використовується оператор `throws` у мові програмування, і яку роль він виконує?
4. Наведіть приклади класів виключень у мові програмування та поясніть їхню призначеність.
5. Який клас виступає на вершині ієрархії класів виключень у Java?
6. Як можна створити користувацький клас виключення в мові програмування Java?
7. Які методи доступні у класі `Throwable` в мові програмування Java, і яку роль вони виконують в системі обробки виключень?

The END
Lec 8.

.NET та JAVA #9

Класи стандартної бібліотеки (Додаткові класи)

1. Стандартна бібліотека Java
2. Математичні обчислення і клас Math
3. Великі числа BigInteger та BigDecimal
4. Робота з датами
5. Класи обгортки

Стандартна бібліотека Java

Стандартна бібліотека Java (The Java Standard Library), також відома як бібліотека класів Java або Java API - це набір класів, інтерфейсів і методів, які постачаються з **JDK**. Ці класи та методи згруповані у пакети, які можна імпортувати до Java-додатків. Стандартна бібліотека призначена для забезпечення широкого спектру функціональних можливостей, від базових операцій, таких як читання і запис файлів, до більш складних функцій, таких як багатопотоковість і мережевий зв'язок.

Нижче перелічено деякі додаткові класи та пакети, доступні в стандартній бібліотеці Java:

1) **java.util**:

- ArrayList: Реалізація динамічного масиву.
- HashMap: Реалізація хеш-таблиці для зберігання ключ-значення пар;
- LinkedList: Пов'язаний список;
- HashSet: Реалізація множини на основі хеш-таблиці;
- TreeMap: Реалізація відсортованої карти;
- Queue і Deque: Інтерфейси для роботи з чергами і двосторонніми чергами.

2) **java.io**:

- File: Клас для роботи з файлами та директоріями;
- FileInputStream і FileOutputStream: Класи для читання і запису файлів;
- BufferedReader і BufferedWriter: Класи для буферизованого читання і запису даних;
- ObjectInputStream і ObjectOutputStream: Класи для серіалізації та десеріалізації об'єктів.

Стандартна бібліотека Java

3) **java.nio:**

- ByteBuffer: Клас для роботи з буферами байтів;
- FileChannel: Клас для роботи з каналами файлів.

4) **java.net:**

- Socket і ServerSocket: Класи для роботи з сокетами і мережевими з'єднаннями;
- URL і URLConnection: Класи для роботи з URL і HTTP-з'єднаннями.

5) **java.util.concurrent:**

- Executor і ExecutorService: Інтерфейси для виконання завдань у багатопотоковому середовищі;
- ThreadPoolExecutor: Реалізація пулу потоків.

6) **java.time:**

- LocalDate, LocalTime, LocalDateTime: Класи для роботи з датами і часом;
- Duration і Period: Класи для роботи з інтервалами часу та періодами.

7) **java.util.regex:**

- Pattern і Matcher: Класи для роботи з регулярними виразами.

8) **java.lang:**

- Math: Клас для математичних операцій.
- StringBuilder, StringBuffer: Клас для побудови змінюваних рядків;
- Integer, Boolean, Character...: Класи обгортки.

Більшість цих додаткових класів ми вже розглядали, але потрібно розібрати деякі з них на цій лекції

Математичні обчислення і клас Math

Для виконання різних математичних операцій у Java в пакеті `java.lang` визначено клас `Math`. Розглянемо його основні методи:

- **`abs(double value)`**: повертає абсолютне значення для аргументу `value`;
- **`acos(double value)`**: повертає арккосинус `value`. Параметр `value` повинен мати значення від -1 до 1;
- **`asin(double value)`**: повертає арксинус `value`. Параметр `value` повинен мати значення від -1 до 1;
- **`atan(double value)`**: повертає арктангенс `value`;
- **`cbrt(double value)`**: повертає кубічний корінь числа `value`;
- **`ceil(double value)`**: повертає найменше ціле число з плаваючою крапкою, яке не менше `value`;
- **`cos(double d)`**: повертає косинус кута;
- **`cosh(double d)`**: повертає гіперболічний косинус кута;
- **`exp(double d)`**: повертає основу натурального логарифма, піднесену до степеня `d`;
- **`floor(double d)`**: повертає найбільше ціле число, яке не більше `d`;
- **`floorDiv(int a, int b)`**: повертає цілочисельний результат ділення `a` на `b`;
- **`log(double a)`**: повертає натуральний логарифм числа `a`;
- **`log1p(double d)`**: повертає натуральний логарифм числа $(d + 1)$;
- **`log10(double d)`**: повертає десятковий логарифм числа `d`;
- **`max(double a, double b)`**: повертає максимальне число з `a` і `b`;
- **`min(double a, double b)`**: повертає мінімальне число з `a` і `b`;
- **`pow(double a, double b)`**: повертає число `a`, зведене до степеня `b`;
- **`random()`**: повертає випадкове число від 0.0 до 1.0;

Математичні обчислення і клас Math

- **round(double d):** повертає число d, округлене до найближчого цілого числа;
- **scalb(double value, int factor):** повертає добуток числа value на 2 у ступені factor;
- **signum(double value):** повертає число 1, якщо число value позитивне, і -1, якщо значення value негативне. Якщо value дорівнює 0, то повертає 0;
- **sin(double value):** повертає синус кута
- **sinh(double value):** повертає гіперболічний синус кута;
- **sqrt(double value):** повертає квадратний корінь числа value;
- **tan(double value):** повертає тангенс кута value;
- **tanh(double value):** повертає гіперболічний тангенс кута value;
- **toDegrees(double value)** переводить радіани в градуси і **toRadians(double value)** - градуси в радіани.

Також клас Math визначає дві константи: Math.E і Math.PI.

Математичні обчислення і клас Math

```
double result = Math.abs(-13.5); // 13.5
result = Math.acos(1); // 0.0
result = Math.asin(5); // NaN
result = Math.atan(5); // 1.3734...
result = Math.cbrt(5); // 1.7099...
result = Math.ceil(6.3); // 7.0
result = Math.cos(15); // -0.7596...
result = Math.cosh(5); // 74.2099
result = Math.exp(5); // 148.41315...
result = Math.floor(5.8); // 5.0
result = Math.floorDiv(6, 4); // 1.0
result = Math.log(5); // 1.6094...
result = Math.log1p(5); // 1.7917...
result = Math.log10(5); // 0.6989...
result = Math.max(5, 3); // 5.0
```

```
result = Math.min(5, 3); // 3.0
result = Math.pow(5, 3); // 125.0
result = Math.random(); // 0.753000...
result = Math rint(10.2); // 10.0
result = Math.round(2.3); // 2.0
result = Math.scalb(5, 3); // 40.0
result = Math.signum(2.3); // 1.0
result = Math.sin(10.5); // -0.8796...
result = Math.sinh(10.5); // 18157.7513...
result = Math.sqrt(10.5); // 3.2403...
result = Math.tan(10.5); // 1.8498...
result = Math.tanh(10.5); // 0.9999...
result = Math.toDegrees(3.14157); // 179.9987...
result = Math.toRadians(90); // 1.57079....
```

Великі числа **BigInteger** та **BigDecimal**

Вбудовані примітивні числові типи не завжди можуть підходити для певних програм. Наприклад, необхідно зберігати і використовувати в програмі дуже великі числа, які виходять за межі допустимих значень для типів `long` і `double`.

У цьому випадку для роботи з числовими даними можна використовувати два додаткові типи з пакета **java.math** - **BigInteger** (для цілочисельних даних) і **BigDecimal** (для чисел з плаваючою крапкою).

Основні методи класу **BigInteger**:

- **BigInteger add(BigInteger other):** повертає суму двох чисел;
- **BigInteger subtract(BigInteger other):** повертає різницю двох чисел;
- **BigInteger multiply(BigInteger other):** повертає добуток двох чисел;
- **BigInteger divide(BigInteger other):** повертає частку двох чисел;
- **BigInteger mod(BigInteger other):** повертає залишок від цілочисельного ділення двох чисел;
- **BigInteger sqrt():** повертає квадратний корінь числа;
- **int compareTo(BigInteger other):** порівнює два числа. Повертає -1, якщо поточний об'єкт менший за число `other`, 1 - якщо поточний об'єкт більший і 0 - якщо числа рівні;
- **static BigInteger valueOf(long x):** повертає об'єкт **BigInteger**, значення якого дорівнює числу, переданому як параметр;
- **int intValue():** конвертує об'єкт **BigInteger** в об'єкт `int`;
- **byte byteValue():** перетворює об'єкт **BigInteger** у `byte`;
- **short shortValue():** перетворює об'єкт **BigInteger** у `short`;
- **long longValue():** перетворює об'єкт **BigInteger** у `long`.

Великі числа BigInteger та BigDecimal

Основні методи класу BigDecimal:

- **BigDecimal add(BigDecimal other):** повертає суму двох чисел;
- **BigDecimal subtract(BigDecimal other):** повертає різницю двох чисел;
- **BigDecimal multiply(BigDecimal other):** повертає добуток двох чисел;
- **BigDecimal divide(BigDecimal other):** повертає частку двох чисел;
- **BigDecimal divide(BigDecimal other, RoundingMode mode):** результат ділення двох чисел, округлений відповідно до режиму mode;
- **int compareTo(BigDecimal other):** порівнює два числа. Повертає -1, якщо поточний об'єкт менший за число other, 1 - якщо поточний об'єкт більший і 0 - якщо числа рівні;
- **static BigDecimal valueOf(double x):** повертає об'єкт BigDecimal, значення якого дорівнює числу, переданому як параметр;
- **double doubleValue():** перетворює об'єкт BigDecimal у double;
- **float floatValue():** перетворює об'єкт BigDecimal на float.

Великі числа BigInteger та BigDecimal

Приклад використання класів **BigInteger** і **BigDecimal**:

```
BigInteger a = BigInteger.valueOf(2147483647);
BigInteger b = BigInteger.valueOf(2147483641);
//a = a * b;  // неможливо
a = a.multiply(b);
System.out.println(a);  // 4611686001247518727
long x = a.longValue();
System.out.println(x);  // 4611686001247518727

BigDecimal c = BigDecimal.valueOf(2325.06);
BigDecimal d = BigDecimal.valueOf(215.06);
c = c.subtract(d.multiply(BigDecimal.valueOf(2.1)));
System.out.println(c);  // 1873.434
double y = c.doubleValue();
System.out.println(y);  // 1873.434
```

Робота з датами. `LocalDate`

Клас **`LocalDate`** з пакета `java.time` призначений для роботи з датами. Функціонал цього класу дає змогу створювати дати і змінювати їх, додаючи і віднімаючи необхідну кількість днів/місяців/років.

Основні методи `LocalDate`:

- **`static LocalDate now()`**: повертає об'єкт, який представляє поточну дату;
- **`static LocalDate of(int year, int month, int day)`**: повертає об'єкт, який представляє дату з визначеними роком, місяцем і днем;
- **`int getYear()`**: повертає рік дати;
- **`int getMonthValue()`**: повертає місяць;
- **`int getDayOfMonth()`**: повертає день місяця (значення від 1 до 31);
- **`int getDayOfYear()`**: повертає номер дня року (значення від 1 до 365);
- **`DayOfWeek getDayOfWeek()`**: повертає день тижня у вигляді значення перерахування `DayOfWeek`;
- **`LocalDate plusDays(int n)`**: додає до дати деяку кількість днів;
- **`LocalDate plusWeeks(int n)`**: додає до дати деяку кількість тижнів;
- **`LocalDate plusMonths(int n)`**: додає до дати деяку кількість місяців;
- **`LocalDate plusYears(int n)`**: додає до дати деяку кількість років;
- **`LocalDate minusDays(int n)`**: віднімає від дати деяку кількість днів;
- **`LocalDate minusMonths(int n)`**: віднімає від дати деяку кількість місяців;
- **`LocalDate minusWeeks(int n)`**: віднімає від дати деяку кількість тижнів;
- **`LocalDate minusYears(int n)`**: віднімає від дати деяку кількість років.

Робота з датами. LocalDate

Приклад використання:

```
LocalDate date = LocalDate.now(); // отримуємо поточну дату
int year = date.getYear();
int month = date.getMonthValue();
int dayOfMonth = date.getDayOfMonth();
DayOfWeek dayOfWeek = date.getDayOfWeek();
System.out.println(date);
System.out.println(dayOfWeek);
System.out.printf("%d.%d.%d \n", dayOfMonth, month, year);
```

```
LocalDate date = LocalDate.of( year: 1914, month: 7, dayOfMonth: 28);
date = date.plusYears( yearsToAdd: 4);
date = date.plusMonths( monthsToAdd: 3);
date = date.plusDays( daysToAdd: 14);
System.out.println(date); // 1918-11-11

date = date.minusMonths( monthsToSubtract: 10);
date = date.minusDays( daysToSubtract: 3);
System.out.println(date); // 1918-01-08
```


Класи обгортки

Клас-обгортка (wrapper class) - це клас, об'єкт якого обгортає або містить примітивні типи даних. Коли ми створюємо об'єкт класу-обгортки, він містить поле, і в цьому полі ми можемо зберігати примітивні типи даних. Іншими словами, ми можемо обернути примітивне значення в об'єкт класу-обгортки. Кожен примітив Java має відповідну обгортку:

- **boolean, byte, short, char, int, long, float, double;**
- **Boolean, Byte, Short, Character, Integer, Long, Float, Double.**

«Навіщо потрібні класи-обгортки?»

Існують певні потреби у використанні класів-обгортки у Java, як зазначено нижче:

- Вони перетворюють примітивні типи даних в об'єкти. Об'єкти потрібні, якщо ми хочемо змінити аргументи, що передаються в метод (оскільки примітивні типи передаються за значенням);
- Класи в пакеті **java.util** працюють тільки з об'єктами, а отже, класи-обгортки допомагають і в цьому випадку;
- Структури даних у фреймворку **Collection**, такі як **ArrayList** та **Vector**, зберігають тільки об'єкти (посилальні типи), а не примітивні типи;
- Об'єкт потрібен для підтримки синхронізації в багатопоточності.

Класи обгортки

Переваги класів-обгорток:

- Колекції допускають тільки об'єктні дані;
- На об'єктних даних можна викликати декілька методів `compareTo()`, `equals()`, `toString()`;
- Клонувати можна тільки об'єкти;
- Об'єктні дані допускають нульові значення;
- Серіалізація може допускати тільки об'єктні дані.

Autoboxing та Unboxing

1. **Autoboxing**. Автоматичне перетворення примітивних типів до об'єктів відповідних класів-обгортки називається автобоксингом. Наприклад, перетворення `int` в **Integer**, `long` в **Long**, `double` в **Double** тощо.

```
char ch = 'a';  
// Автобоксинг - перетворення примітивів у символні об'єкти  
Character a = ch;  
ArrayList<Integer> arrayList = new ArrayList<Integer>();  
  
// Автобоксинг, оскільки ArrayList зберігає лише об'єкти  
arrayList.add(25);  
System.out.println(arrayList.get(0));
```

Класи обгортки

2. **Unboxing**. Це зворотний процес до autoboxing. Автоматичне перетворення об'єкта класу-обгортки до відповідного примітивного типу називається розпаковуванням. Наприклад, перетворення **Integer** в **int**, **Long** в **long**, **Double** в **double** тощо.

```
Character ch = 'a';  
// Unboxing - перетворення символічного об'єкта на примітив  
char a = ch;  
  
ArrayList<Integer> arrayList = new ArrayList<Integer>();  
arrayList.add(24);  
  
// розпакування, тому що метод get повертає Integer об'єкт  
int num = arrayList.get(0);  
System.out.println(num);
```

Класи обгортки

Приклад класів обгортки Java

```
public static void main(String[] args) {  
    byte a = 1;  
    Byte byteobj = a;  
  
    int b = 10;  
    Integer intobj = b;  
  
    float c = 18.6f;  
    Float floatobj = c;  
  
    double d = 250.5;  
    Double doubleobj = d;  
  
    char e = 'a';  
    Character charobj = e;  
  
    byte bv = byteobj;  
    int iv = intobj;  
    float fv = floatobj;  
    double dv = doubleobj;  
    char cv = charobj;  
}
```

Контрольні питання

1. Які ви знаєте додаткові класи та пакети, доступні в стандартній бібліотеці Java?
2. Який конкретний клас використовується для виконання математичних обчислень у мові програмування Java?
3. Які класи можна використати, якщо числа з якими ви працюєте виходять за межі допустимих значень для типів long і double?
4. За допомогою якого класу можна працювати з датами в мові програмування Java?
5. Охарактеризуйте поняття "класи-обгортки" у контексті мови програмування Java та поясніть їхню призначеність.

The END
Lec 9.

.NET та JAVA #10

Робота з файловою системою та потоками введення-виведення. Серіалізація об'єктів.

1. Ознайомлення з потоками введення-виведення
2. Читання та запис файлів
3. Буферізовані потоки
4. Форматоване введення та виведення
5. Читання та запис примітивних типів даних
6. Читання та запис текстових файлів
7. Буферізація символьних потоків
8. Серіалізація об'єктів
9. Клас File. Робота з файлами та каталогами
10. Робота з ZIP-архівами

Ознайомлення з потоками введення-виведення

Відмінною рисою багатьох мов програмування є робота з **файлами** і **потоками**. У Java основний функціонал роботи з потоками зосереджений у класах із пакета **java.io**.

Ключовим поняттям тут є поняття **потoku (stream)**. Хоча поняття "**потік**" у програмуванні доволі перевантажене і може позначати безліч різних концепцій. У цьому разі стосовно роботи з файлами і введенням-виведенням ми говоритимемо про потік (stream), як про абстракцію, що використовується для читання або запису інформації (файлів, сокетів, тексту консолі тощо).

Ознайомлення з потоками введення-виведення

Потік пов'язаний з реальним фізичним пристроєм за допомогою системи введення-виведення **Java**. У нас може бути визначений потік, який пов'язаний з файлом і через який ми можемо вести читання або запис файлу. Це також може бути потік, пов'язаний із мережевим сокетом, за допомогою якого можна отримати або надіслати дані в мережі.

Усі ці завдання: читання і запис різних файлів, обмін інформацією мережею, введення-виведення в консолі ми будемо вирішувати в Java за допомогою **потоків**.

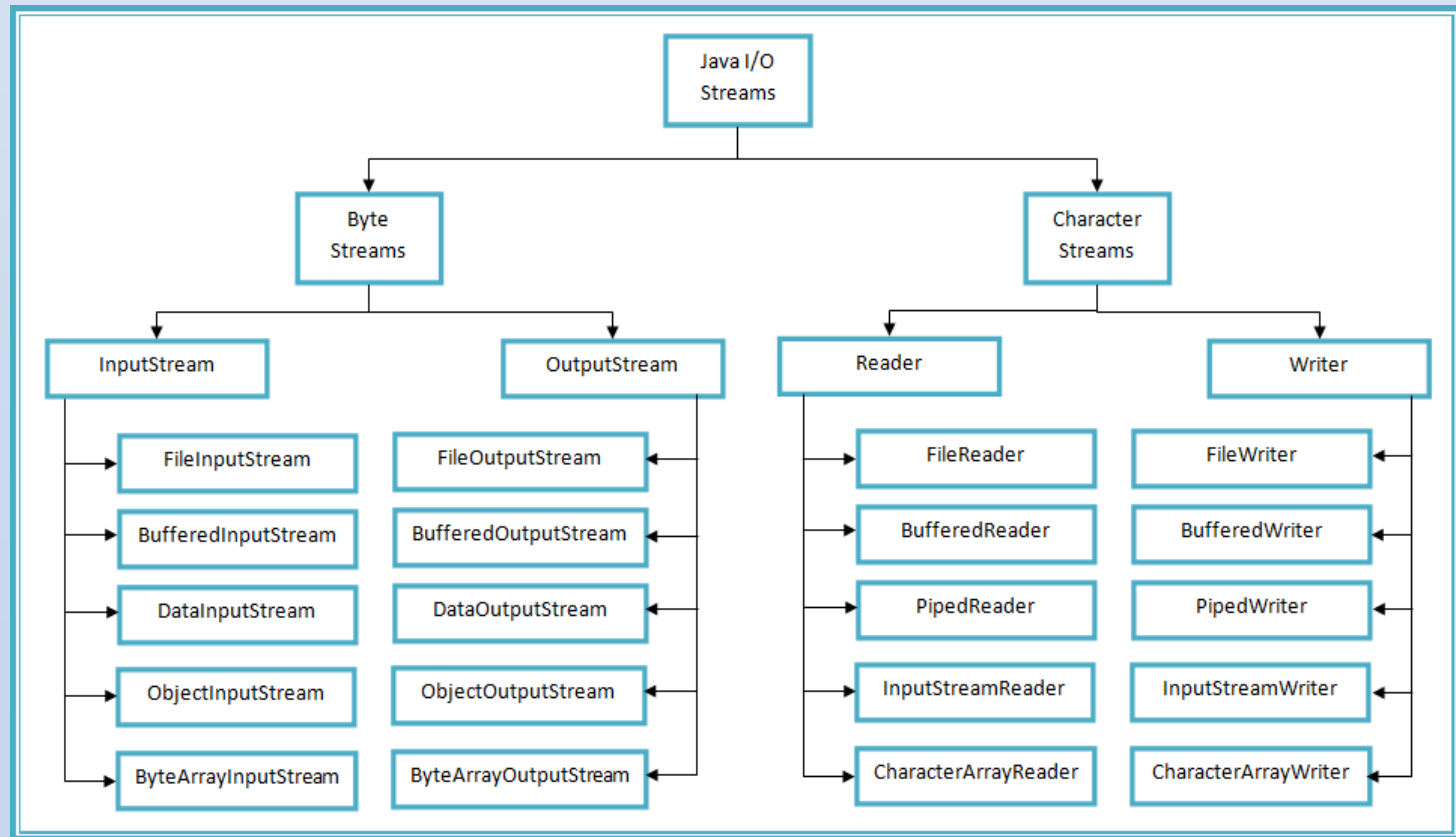
Об'єкт, з якого можна зчитати дані, називається **потоком введення**, а об'єкт, у який можна записувати дані, - **потоком виведення**. Наприклад, якщо треба зчитати вміст файлу, то застосовується потік введення, а якщо треба записати у файл - то потік виведення.

В основі всіх класів, що керують потоками байтів, лежать два абстрактні класи: **InputStream** (що представляє потоки введення) і **OutputStream** (що представляє потоки виведення).

Але оскільки працювати з байтами не дуже зручно, то для роботи з потоками символів були додані абстрактні класи **Reader** (для читання потоків символів) і **Writer** (для запису потоків символів).

Ознайомлення з потоками введення-виведення

Усі інші класи, що працюють із потоками, є спадкоємцями цих абстрактних класів. Основні класи потоків:



Ознайомлення з потоками введення-виведення

Потоки байтів

Клас **InputStream** – базовий клас для всіх класів, які управляють байтовими потоками введення. Основні методи:

- **int available():** повертає кількість байтів, доступних для читання в потоці;
- **void close():** закриває потік;
- **int read():** повертає цілочисельне представлення наступного байта в потоці. Коли в потоці не залишиться доступних для читання байтів, цей метод поверне число -1;
- **int read(byte[] buffer):** зчитує байти з потоку в масив buffer. Після читання повертає число зчитаних байтів. Якщо жодного байта не було зчитано, то повертається число -1;
- **int read(byte[] buffer, int offset, int length):** зчитує деяку кількість байтів, що дорівнює length, з потоку в масив buffer. При цьому зчитані байти поміщаються в масиві, починаючи зі зміщення offset, тобто з елемента buffer[offset]. Метод повертає число успішно прочитаних байтів;
- **long skip(long number):** пропускає в потоці під час читання деяку кількість байт, яка дорівнює number.

Ознайомлення з потоками введення-виведення

Клас **OutputStream** – базовий клас для всіх класів, які працюють з байтовими потоками виведення. Основні методи:

- **void close():** закриває потік;
- **void flush():** очищає буфер виведення, записуючи весь його вміст;
- **void write(int b):** записує у вихідний потік один байт, який представлений цілочисельним параметром b **void write(byte[] buffer):** записує у вихідний потік масив байтів buffer;
- **void write(byte[] buffer, int offset, int length):** записує у вихідний потік деяку кількість байтів, що дорівнює length, з масиву buffer, починаючи зі зміщення offset, тобто з елемента buffer[offset].

Класи символьних потоків

Символьні потоки мають два основних абстрактних класи **Reader** і **Writer**, що керують потоками символів Unicode.

Основні методи абстрактного класу **Reader**:

- **abstract void close():** закриває потік введення;
- **int read():** повертає цілочисельне представлення наступного символу в потоці. Якщо таких символів немає, і досягнуто кінця файлу, то повертається число -1;
- **int read(char[] buffer):** зчитує в масив buffer з потоку символи, кількість яких дорівнює довжині масиву buffer. Повертає кількість успішно зчитаних символів. При досягненні кінця файлу повертає -1;

Ознайомлення з потоками введення-виведення

- **int read(CharBuffer buffer):** зчитує в об'єкт CharBuffer з потоку символи. Повертає кількість успішно зчитаних символів. При досягненні кінця файлу повертає -1;
- **abstract int read(char[] buffer, int offset, int count):** зчитує в масив buffer, починаючи зі зміщення offset, з потоку символи, кількість яких дорівнює count;
- **long skip(long count):** пропускає кількість символів, що дорівнює count. Повертає число успішно пропущених символів.

Основні методи абстрактного класу **Writer**:

- **Writer append(char c):** додає в кінець вихідного потоку символ c. Повертає об'єкт Writer;
- **Writer append(CharSequence chars):** додає в кінець вихідного потоку набір символів chars. Повертає об'єкт Writer;
- **abstract void close():** закриває потік;
- **abstract void flush():** очищає буфери потоку;
- **void write(int c):** записує в потік один символ, який має цілочисельне представлення;
- **void write(char[] buffer):** записує в потік масив символів;
- **abstract void write(char[] buffer, int off, int len) :** записує в потік тільки кілька символів з масиву buffer. Причому кількість символів дорівнює len, а відбір символів з масиву починається з індексу off;

Ознайомлення з потоками введення-виведення

- **void write(String str):** записує в потік рядок;
- **void write(String str, int off, int len):** записує в потік із рядка деяку кількість символів, що дорівнює len, причому відбір символів із рядка починається з індексу off.

Функціонал, описаний класами **Reader** і **Writer**, успадковується безпосередньо класами символьних потоків, зокрема класами **FileReader** і **FileWriter** відповідно, призначеними для роботи з текстовими файлами.

Читання та запис файлів

Запис файлів і клас *FileOutputStream*

Клас **FileOutputStream** призначений для запису байтів у файл. Він є похідним від класу **OutputStream**, тому успадковує всю його функціональність. Через конструктор класу **FileOutputStream** задається файл, у який здійснюється запис. Клас підтримує кілька конструкторів:

```
FileOutputStream(String filePath)
```

```
FileOutputStream(File fileObj)
```

```
FileOutputStream(String filePath, boolean append)
```

```
FileOutputStream(File fileObj, boolean append)
```

Через перший параметр задається шлях до файлу. Другий параметр - **append** задає спосіб запису: **true** - дозапис в кінець файлу, **false** - файл повністю перезаписується.

Читання та запис файлів

```
class Program {  
    public static void main(String[] args) {  
        String text = "Hello world!"; // рядок для запису  
        try(FileOutputStream fos=new FileOutputStream( name: "notes.txt")){  
            // конвертуємо рядок в масив байтів  
            byte[] buffer = text.getBytes();  
  
            fos.write(buffer, off: 0, buffer.length);  
            System.out.println("The file has been written");  
        }  
        catch(IOException ex){  
            System.out.println(ex.getMessage());  
        }  
    }  
}
```

Для створення об'єкта **FileOutputStream** використовується конструктор, що приймає як параметр шлях до файлу для запису. Якщо такого файлу немає, то він автоматично створюється під час запису. Оскільки тут записуємо рядок, то його треба спочатку перевести в масив байтів. І за допомогою методу write рядок записується у файл.

Читання та запис файлів

Для автоматичного закриття файлу і звільнення ресурсу об'єкт **FileOutputStream** створюється за допомогою конструкції try...catch. (**try-with-resources**)

Читання файлів і клас **FileInputStream**

Для зчитування даних з файлу призначений клас **FileInputStream**, який є спадкоємцем класу **InputStream** і тому реалізує всі його методи.

Для створення об'єкта **FileInputStream** ми можемо використовувати низку конструкторів. Найбільш використовувана версія конструктора, коли в якості параметру приймається шлях до зчитуваного файлу:

FileInputStream(String fileName) throws FileNotFoundException

Якщо файл не може бути відкритий, наприклад, за вказаним шляхом такого файлу не існує, то генерується виняток **FileNotFoundException**. Наприклад, зчитуємо дані з раніше записаного файлу і виведемо на консоль:

```
public static void main(String[] args) {  
    try(FileInputStream fin=new FileInputStream("name: notes.txt")){  
        int i;  
        while((i=fin.read())!=-1)  
            System.out.print((char)i);  
    }  
    catch(IOException ex){  
        System.out.println(ex.getMessage());  
    }  
}
```

У цьому випадку ми зчитуємо кожен окремий байт у змінну i, коли в потоці більше немає даних для читання, метод повертає число -1. Потім кожен зчитаний байт конвертується в об'єкт типу char і виводиться на консоль.

Читання та запис файлів

Подібним чином можна зчитати дані в масив байтів і потім проводити з ним маніпуляції:

```
try(FileInputStream fin=new FileInputStream( name: "notes.txt")){
    byte[] buffer = new byte[256];
    System.out.println("File data:");
    int count;
    while((count=fin.read(buffer))!=-1)
        for(int i=0; i<count;i++)
            System.out.print((char)buffer[i]);
}
catch(IOException ex){
    System.out.println(ex.getMessage());
}
```

У цьому випадку за допомогою методу **read()** зчитуємо дані в масив `buffer` довжиною 256 байтів. Метод повертає кількість зчитаних байтів. Оскільки файл може бути більшим за 256 байтів, то зчитуємо в циклі `while` до кінця файлу. Коли більше не залишиться файлів для зчитування, то метод поверне -1.

Читання та запис файлів

Сумістимо обидва класи і виконаємо читання з одного і запис в інший файл:

```
try(FileInputStream fin=new FileInputStream( name: "notes.txt");
    FileOutputStream fos=new FileOutputStream( name: "notes_new.txt")){
    byte[] buffer = new byte[256];
    int count;
    // зчитуємо бUFFER
    while((count=fin.read(buffer))!=-1)
    {
        // записуємо з бUFFERа у файл
        fos.write(buffer, off: 0, count);
        System.out.println("File has been written");
    }
}
catch(IOException ex){
    System.out.println(ex.getMessage());
}
```

Класи **FileInputStream** і **FileOutputStream** призначені насамперед для запису двійкових файлів, тобто для запису і читання байтів. І хоча вони також можуть використовуватися для роботи з текстовими файлами, але все ж для цього завдання більше підходять інші класи.

Читання та запис байтових масивів

Для роботи з масивами байтів - їхнього читання і запису використовуються класи **ByteArrayInputStream** і **ByteArrayOutputStream**.

Читання масиву байтів і клас **ByteArrayInputStream**

Клас **ByteArrayInputStream** представляє вхідний потік, який використовує як джерело даних масив байтів. Він має такі конструктори:

ByteArrayInputStream(byte[] buf)

ByteArrayInputStream(byte[] buf, int offset, int length)

Як параметри конструктори використовують масив байтів **buf**, з якого здійснюється зчитування, зсув щодо початку масиву **offset** і кількість символів, що зчитуються, **length**.

Зчитуємо масив байтів і виведемо його на екран:

```
public static void main(String[] args) {
    byte[] array1 = new byte[]{1, 3, 5, 7};
    ByteArrayInputStream byteStream1 = new ByteArrayInputStream(array1);
    int b;
    while((b=byteStream1.read())!=-1)
        System.out.println(b);

    String text = "Hello world!";
    byte[] array2 = text.getBytes();
    // отримуємо 5 символів
    ByteArrayInputStream byteStream2 = new ByteArrayInputStream(array2, offset: 0, length: 5);
    int c;
    while((c=byteStream2.read())!=-1)
        System.out.println((char)c);
}
```

На відміну від інших класів потоків для закриття об'єкта **ByteArrayInputStream** не потрібно викликати метод **close**.

Читання та запис байтових масивів

Запис масиву байт та клас `ByteArrayOutputStream`

Клас **`ByteArrayOutputStream`** представляє потік виведення, що використовує масив байтів як місце виведення. Щоб створити об'єкт цього класу, ми можемо використовувати один з його конструкторів:

`ByteArrayOutputStream()`

`ByteArrayOutputStream(int size)`

Перша версія створює масив для зберігання байтів завдовжки 32 байти, а друга версія створює масив довжиною `size`.

```
ByteArrayOutputStream out = new ByteArrayOutputStream();
String text = "Hello World!";
byte[] buffer = text.getBytes();
try{
    out.write(buffer);
}
catch(Exception ex){
    System.out.println(ex.getMessage());
}
// перетворюємо масив байтів на рядок
System.out.println(out);

// отримуємо масив байтів і виводимо по символічно
byte[] array = out.toByteArray();
for(byte b: array)
    System.out.print((char)b);

System.out.println();
```

Як і в інших потоках виведення, у класі **`ByteArrayOutputStream`** визначено метод `write`, який записує в потік деякі дані. У цьому випадку ми записуємо в потік масив байтів. Цей масив байтів записується в об'єкті **`ByteArrayOutputStream`** у захищене поле `buf`, яке представляє також масив байтів(`protected byte[] buf`).

Читання та запис байтових масивів

```
ByteArrayOutputStream out = new ByteArrayOutputStream();
String text = "Hello World!";
byte[] buffer = text.getBytes();
try{
    out.write(buffer);
}
catch(Exception ex){
    System.out.println(ex.getMessage());
}
// перетворюємо масив байтів на рядок
System.out.println(out);

// отримуємо масив байтів і виводимо по символічно
byte[] array = out.toByteArray();
for(byte b: array)
    System.out.print((char)b);

System.out.println();
```

Як і в інших потоках виведення, у класі **ByteArrayOutputStream** визначено метод `write`, який записує в потік деякі дані. У цьому випадку ми записуємо в потік масив байтів. Цей масив байтів записується в об'єкті **ByteArrayOutputStream** у захищене поле `buf`, яке представляє також масив байтів (`protected byte[] buf`).

Читання та запис байтових масивів

Оскільки метод **write** може згенерувати виняток, то виклик цього методу поміщається в блок `try..catch`. Використовуючи методи `toString()` і `toByteArray()`, можна отримати масив байтів `buf` у вигляді тексту або безпосередньо у вигляді масиву байт.

За допомогою методу `writeTo()` ми можемо вивести масив байт в інший потік. Цей метод як параметр приймає об'єкт **OutputStream**, у який здійснюється запис масиву байт:

```
ByteArrayOutputStream out = new ByteArrayOutputStream();
String text = "Hello World!";
byte[] buffer = text.getBytes();
try{
    out.write(buffer);
}
catch(Exception ex){
    System.out.println(ex.getMessage());
}
try(FileOutputStream fos = new FileOutputStream( name: "hello.txt")){
    out.writeTo(fos);
}
catch(IOException e){
    System.out.println(e.getMessage());
}
```

Буферізовані потоки

Для оптимізації операцій введення-виведення використовуються **буферізовані потоки**. Ці потоки додають до стандартних спеціальний буфер у пам'яті, за допомогою якого підвищується продуктивність під час читання і запису потоків.

Клас **BufferedInputStream**

BufferedInputStream накопичує дані, що вводяться, у спеціальному буфері без постійного звернення до пристрою введення. Клас **BufferedInputStream** визначає два конструктори:

BufferedInputStream(InputStream inputStream)

BufferedInputStream(InputStream inputStream, int bufSize)

Перший параметр - це потік введення, з якого дані будуть зчитуватися в буфер. Другий параметр - розмір буфера.

```
String text = "Hello world!";
byte[] buffer = text.getBytes();
ByteArrayInputStream in = new ByteArrayInputStream(buffer);

try(BufferedInputStream out = new BufferedInputStream(in)){
    int c;
    while((c=out.read())!=-1)
        System.out.print((char)c);
}
catch(Exception e){
    System.out.println(e.getMessage());
}
System.out.println();
```

Клас **BufferedInputStream** у конструкторі приймає об'єкт **InputStream**. У цьому випадку таким об'єктом є екземпляр класу **ByteArrayInputStream**. Як і всі потоки введення **BufferedInputStream** має метод **read()**, який зчитує дані. І тут ми зчитуємо за допомогою методу **read** кожен байт з масиву **buffer**.

Буферізовані потоки

Фактично все те ж саме можна було зробити і за допомогою одного **ByteArrayInputStream**, не вдаючись до буферизованого потоку. Клас **BufferedInputStream** просто оптимізує продуктивність при роботі з потоком **ByteArrayInputStream**. Природно замість **ByteArrayInputStream** може використовуватися будь-який інший клас, який успадкований від **InputStream**.

BufferedOutputStream(OutputStream outputStream)

BufferedOutputStream(OutputStream outputStream, int bufSize)

Перший параметр - це потік виведення, який успадкований від **OutputStream**, а другий параметр - розмір буфера.

Приклад запису до файлу:

```
String text = "Hello world!"; // рядок для запису
try(FileOutputStream out=new FileOutputStream( name: "notes.txt");
    BufferedOutputStream bos = new BufferedOutputStream(out)){
    // переведення рядка в байти
    byte[] buffer = text.getBytes();
    bos.write(buffer, off: 0, buffer.length);
}
catch(IOException ex){
    System.out.println(ex.getMessage());
}
```

Клас **BufferedOutputStream** у конструкторі приймає як параметр об'єкт **OutputStream** - у даному випадку це файловий потік виведення **FileOutputStream**. І також здійснюється запис у файл. Знову ж **BufferedOutputStream** не додає багато нової функціональності, він просто оптимізує дію потоку виведення.

Форматоване введення та виведення

Клас **PrintStream**

Клас **PrintStream** - це саме той клас, який використовується для виведення на консоль. Коли ми виводимо на консоль деяку інформацію за допомогою виклику `System.out.println()`, то тим самим ми задіємо **PrintStream**, тому що змінна `out` у класі `System` якраз і представляє об'єкт класу **PrintStream**, а метод `println()` - це метод класу **PrintStream**.

Але **PrintStream** корисний не тільки для виведення на консоль. Ми можемо використовувати цей клас для запису інформації в потік виведення. Для цього **PrintStream** визначає низку конструкторів:

```
PrintStream(OutputStream outputStream)
```

```
PrintStream(OutputStream outputStream, boolean autoFlushingOn)
```

```
PrintStream(OutputStream outputStream, boolean autoFlushingOn, String  
charSet) throws UnsupportedOperationException
```

```
PrintStream(File outputFile) throws FileNotFoundException
```

```
PrintStream(File outputFile, String charSet) throws FileNotFoundException,  
UnsupportedEncodingException
```

```
PrintStream(String outputFileName) throws FileNotFoundException
```

```
PrintStream(String outputFileName, String charSet) throws  
FileNotFoundException, UnsupportedOperationException
```

Форматоване введення та виведення

Параметр **outputStream** - це об'єкт **OutputStream**, у який здійснюється запис. Параметр **autoFlushingOn** при значенні true дає змогу автоматично записувати дані в потік виведення. За замовчуванням цей параметр дорівнює false. Параметр **charSet** дає змогу вказати кодування символів. Як джерело для запису даних замість **OutputStream** можна використовувати об'єкт **File** або строковий шлях, за яким буде створюватися файл.

Для виведення інформації у вихідний потік **PrintStream** використовує такі методи:

- **println():** виведення рядкової інформації з переведенням рядка;
- **print():** виведення рядкової інформації без переведення рядка;
- **printf():** форматоване виведення.

```
String text = "Hello, World!"; // рядок для запису
try(FileOutputStream fos=new FileOutputStream( name: "notes3.txt");
    PrintStream printStream = new PrintStream(fos)) {
    printStream.println(text);
    System.out.println("Запис у файл здійснено");
}
catch(IOException ex){
    System.out.println(ex.getMessage());
}
```

У цьому випадку застосовується форма конструктора **PrintStream**, яка як параметр приймає потік виведення: **PrintStream** (OutputStream out). Крім того, ми могли б використовувати низку інших форм конструктора, наприклад, вказуючи назви файлу для запису: **PrintStream** (string filename)

Форматоване введення та виведення

Як потік виведення використовується об'єкт **FileOutputStream**. За допомогою методу **println()** здійснюється запис інформації у вихідний потік - тобто в об'єкт **FileOutputStream**. (У випадку з виведенням на консоль за допомогою **System.out.println()** як потік виведення виступає консоль).

Крім того, як і будь-який потік виведення і спадкоємець класу **OutputStream** він має метод **write**:

```
try(PrintStream printStream = new PrintStream( fileName: "notes3.txt")){
    printStream.print("Hello World!");
    printStream.println("Welcome to Java!");

    printStream.printf("Name: %s Age: %d \n", "Tom", 34);

    String message = "PrintStream";
    byte[] message_toBytes = message.getBytes();
    printStream.write(message_toBytes);

    System.out.println("The file has been written");
}
catch(IOException ex){
    System.out.println(ex.getMessage());
}
```

Форматоване введення та виведення

Клас **PrintWriter**

На **PrintStream** схожий інший клас **PrintWriter**. Його можна використовувати як для виведення інформації на консоль, так і в файл або будь-який інший потік виведення. Цей клас має низку конструкторів;

- **PrintWriter(File file file):** автоматично додає інформацію у вказаний файл;
- **PrintWriter(File file, String csn):** автоматично додає інформацію в зазначений файл з урахуванням кодування csн;
- **PrintWriter(OutputStream out):** для виведення інформації використовується наявний об'єкт OutputStream, автоматично скидаючи в нього дані;
- **PrintWriter(OutputStream out, boolean autoFlush):** для виведення інформації використовується наявний об'єкт OutputStream, другий параметр вказує, чи треба автоматично додавати в OutputStream дані;
- **PrintWriter(String fileName):** автоматично додає інформацію у файл за вказаним ім'ям;
- **PrintWriter(String fileName, String csн):** автоматично додає інформацію у файл за вказаним ім'ям, використовуючи кодування csн;
- **PrintWriter(Writer out):** для виведення інформації використовується наявний об'єкт Writer, у який автоматично йде запис даних;
- **PrintWriter(Writer out, boolean autoFlush):** для виведення інформації використовується наявний об'єкт Writer, другий параметр вказує, чи потрібно автоматично додавати в Writer дані.

Форматоване введення та виведення

PrintWriter реалізує інтерфейси **Appendable**, **Closable** і **Flushable**, і тому після використання потік, який він представляє, треба закривати. Для запису даних у потік він також використовує методи `printf()` і `println()`. Наприклад, застосуємо цей клас для виведення на консоль:

```
try(PrintWriter pw = new PrintWriter(System.out)){  
pw.println("Hello world!");  
}
```

Як потік виведення тут застосовується **System.out**, а на консоль буде виведено рядок **"Hello world!"**

Читання та запис примітивних типів даних

Класи **DataOutputStream** і **DataInputStream** дають змогу записувати і зчитувати дані примітивних типів.

Запис даних і **DataOutputStream**

Клас **DataOutputStream** представляє потік виведення і призначений для запису даних примітивних типів, таких, як **int**, **double** тощо. Для запису кожного з примітивних типів призначений свій метод:

- **writeBoolean(boolean v)** : записує в потік булеве однобайтове значення;
- **writeByte(int v)**: записує в потік 1 байт, який представлений у вигляді цілочисельного значення;
- **writeChar(int v)**: записує 2-байтове значення **char**;
- **writeDouble(double v)**: записує в потік 8-байтове значення **double**;
- **writeFloat(float v)**: записує в потік 4-байтове значення **float**;
- **writeInt(int v)**: записує в потік цілочисельне значення **int**;
- **writeLong(long v)**: записує в потік значення **long**;
- **writeShort(int v)**: записує в потік значення **short**;
- **writeUTF(String str)**: записує в потік рядок у кодуванні UTF-8.

Читання та запис примітивних типів даних

Зчитування даних і `DataInputStream`

Клас **`DataInputStream`** діє протилежним чином - він зчитує з потоку дані примітивних типів. Відповідно для кожного примітивного типу визначено свій метод для зчитування:

- **`boolean readBoolean()`**: зчитує з потоку булеве одnobайтове значення;
- **`byte readByte()`**: зчитує з потоку 1 байт;
- **`char readChar()`**: зчитує з потоку значення `char`;
- **`double readDouble()`**: зчитує з потоку 8-байтове значення `double`;
- **`float readFloat()`**: зчитує з потоку 4-байтове значення `float`;
- **`int readInt()`**: зчитує з потоку цілочисельне значення `int`;
- **`long readLong()`**: зчитує з потоку значення `long`;
- **`short readShort()`**: зчитує значення `short`.
- **`String readUTF()`**: зчитує з потоку рядок у кодуванні UTF-8;
- **`int skipBytes(int n)`**: пропускає при читанні з потоку `n` байтів.

Розглянемо приклад роботи:

Читання та запис примітивних типів даних

- **String readUTF():** зчитує з потоку рядок у кодуванні UTF-8;
- **int skipBytes(int n):** пропускає при читанні з потоку n байтів.

Розглянемо приклад роботи:

```
Person tom = new Person( n: "Tom", a: 34, h: 1.68, m: false);
// запис до файлу
try(DataOutputStream dos = new DataOutputStream(new FileOutputStream( name: "data.bin"))) {
    // записуємо значення
    dos.writeUTF(tom.name);
    dos.writeInt(tom.age);
    dos.writeDouble(tom.height);
    dos.writeBoolean(tom.married);
    System.out.println("File has been written");
}
catch(IOException ex){
    System.out.println(ex.getMessage());
}

// зворотнє зчитування з файлу
try(DataInputStream dos = new DataInputStream(new FileInputStream( name: "data.bin"))) {
    // записуємо значення
    String name = dos.readUTF();
    int age = dos.readInt();
    double height = dos.readDouble();
    boolean married = dos.readBoolean();
    System.out.printf("Name: %s Age: %d Height: %f Married: %b",
        name, age, height, married);
}
catch(IOException ex){
    System.out.println(ex.getMessage());
}
```

```
class Person {
    2 usages
    public String name;
    2 usages
    public int age;
    2 usages
    public double height;
    2 usages
    public boolean married;
    1 usage
    public Person(String n, int a, double h, boolean m) {
        this.name=n;
        this.height=h;
        this.age=a;
        this.married=m;
    }
}
```

Читання та запис текстових файлів

Тут ми послідовно записуємо у файл дані об'єкта `Person`. Об'єкт **`DataOutputStream`** у конструкторі приймає потік виведення: **`DataOutputStream`** (`OutputStream out`). У цьому випадку як потік виведення використовується об'єкт `FileOutputStream`, тому виведення відбуватиметься у файл. І за допомогою вище розглянутих методів типу `writeUTF()` проводиться запис значень у бінарний файл. Потім відбувається читання раніше записаних даних. Об'єкт **`DataInputStream`** у конструкторі приймає потік для читання: **`DataInputStream`** (`InputStream in`). Тут таким потоком виступає об'єкт **`FileInputStream`**.

Хоча за допомогою раніше розглянутих класів можна записувати текст у файли, однак вони призначені насамперед для роботи з бінарними потоками даних, і їхніх можливостей для повноцінної роботи з текстовими файлами недостатньо. І для цієї мети служать зовсім інші класи, які є спадкоємцями абстрактних класів **`Reader`** і **`Writer`**.

Клас `FileWriter`

`FileWriter` є похідним від класу **`Writer`**. Він використовується для запису текстових файлів. Щоб створити об'єкт **`FileWriter`**, можна використати один із таких конструкторів:

```
FileWriter(File file)
```

```
FileWriter(File file, boolean append)
```

```
FileWriter(FileDescriptor fd)
```

```
FileWriter(String fileName)
```

```
FileWriter(String fileName, boolean append)
```

Читання та запис текстових файлів

Так, у конструктор передається або шлях до файлу у вигляді рядка, або об'єкт **File**, який посилається на конкретний текстовий файл. Параметр `append` вказує, чи повинні дані дозаписуватися в кінець файлу (якщо параметр дорівнює `true`), чи файл має перезаписуватися.

Розглянемо приклад:

```
try(FileWriter writer = new FileWriter( fileName: "notes3.txt", append: false)) {  
    // запис усього рядка  
    String text = "Hello Gold!";  
    writer.write(text);  
    // запис за символами  
    writer.append('\n');  
    writer.append('E');  
    writer.flush();  
}  
catch(IOException ex){  
    System.out.println(ex.getMessage());  
}
```

Читання та запис текстових файлів

У конструкторі використовувався параметр `append` зі значенням `false` - тобто файл буде перезаписуватися. Потім за допомогою методів, визначених у базовому класі **Writer**, здійснюється запис даних.

Читання файлів. Клас **FileReader**

FileReader успадковується від абстрактного класу **Reader** і надає функціональність для читання текстових файлів. Для створення об'єкта **FileReader** ми можемо використовувати один з його конструкторів:

FileReader(String fileName)

FileReader(File file)

FileReader(FileDescriptor fd)

Використовуючи методи, визначені в базовому класі **Reader**, здійснити читання файлу:

```
try(FileReader reader = new FileReader("notes3.txt")) {  
    // читаємо посимвольно  
    int c;  
    while((c=reader.read())!=-1)  
        System.out.print((char)c);  
}  
catch(IOException ex){  
    System.out.println(ex.getMessage());  
}
```

Також ми можемо зчитувати в проміжний буфер із масиву символів:

```
try(FileReader reader = new FileReader("notes3.txt")) {  
    char[] buf = new char[256];  
    int c;  
    while((c = reader.read(buf))>0){  
        if(c < 256)  
            buf = Arrays.copyOf(buf, c);  
        System.out.print(buf);  
    }  
}  
catch(IOException ex){  
    System.out.println(ex.getMessage());  
}
```

Читання та запис текстових файлів

У цьому випадку зчитуємо послідовно символи з файлу в масив із 256 символів, поки не дійдемо до кінця файлу, у цьому випадку метод `read` поверне число `-1`.

Оскільки зчитана порція файлу може бути меншою за 256 символів (наприклад, у файлі всього 73 символи), і якщо кількість зчитаних даних менша за розмір буфера (256), то виконуємо копіювання масиву за допомогою методу `Arrays.copyOf`.

Тобто фактично обрізаємо масив `buf`, залишаючи в ньому тільки ті символи, які зчитані з файлу.

Буферизація символних потоків

Запис тексту через буфер і **BufferedWriter**

Клас **BufferedWriter** записує текст у потік, попередньо буферизуючи записувані символи, тим самим знижуючи кількість звернень до фізичного носія для запису даних. Клас **BufferedWriter** має такі конструктори:

BufferedWriter(Writer out)

BufferedWriter(Writer out, int sz)

Як параметр він приймає потік виведення, у який треба здійснити запис. Другий параметр вказує на розмір буфера. Наприклад, здійснимо запис у файл:

```
try(BufferedWriter bw = new BufferedWriter(new FileWriter("notes4.txt"))) {
    String text = "Hello World!\nHey! Teachers! Leave the kids alone.";
    bw.write(text);
}
catch(IOException ex){
    System.out.println(ex.getMessage());
}
```

Буферизація символних потоків

Читання тексту та **BufferedReader**

Клас **BufferedReader** зчитує текст із символного потоку введення, буферизуючи прочитані символи. Використання буфера покликане збільшити продуктивність читання даних із потоку. Клас **BufferedReader** має такі конструктори:

BufferedReader(Reader in)

BufferedReader(Reader in, int sz)

Другий конструктор, крім потоку введення, з якого проводиться читання, також визначає розмір буфера, в який будуть зчитуватися символи. Оскільки **BufferedReader** успадковується від класу Reader, то він може використовувати всі ті методи для читання з потоку, які визначені в Reader. І також **BufferedReader** визначає свій власний метод `readLine()`, який дає змогу зчитувати з потоку рядково. Розглянемо застосування **BufferedReader**:

```
try(BufferedReader br = new BufferedReader(new FileReader("notes4.txt"))) {  
    //читання посимвольно  
    int c;  
    while((c=br.read())!=-1)  
        System.out.print((char)c);  
}  
catch(IOException ex){  
    System.out.println(ex.getMessage());  
}
```

Також можна зчитати текст рядково:

```
try(BufferedReader br = new BufferedReader(new FileReader("notes4.txt"))) {  
    //читання рядково  
    String s;  
    while((s=br.readLine())!=null)  
        System.out.println(s);  
}  
catch(IOException ex){  
    System.out.println(ex.getMessage());  
}
```

Серіалізація

Серіалізація (Serialization) являє собою процес запису стану об'єкта в потік, відповідно процес вилучення або відновлення стану об'єкта з потоку називається десеріалізацією. **Серіалізація** дуже зручна, коли йде робота зі складними об'єктами.

Інтерфейс **Serializable**

Відразу треба сказати, що серіалізувати можна тільки ті об'єкти, які реалізують інтерфейс **Serializable**. Цей інтерфейс не визначає жодних методів, просто він слугує вказівником системі, що об'єкт, який його реалізує, може бути серіалізований.

Серіалізація. Клас **ObjectOutputStream**

Для серіалізації об'єктів у потік використовується клас **ObjectOutputStream**. Він записує дані в потік. Для створення об'єкта **ObjectOutputStream** у конструктор передається потік, у який здійснюється запис:
ObjectOutputStream(OutputStream out)

Серіалізація

Для запису даних **ObjectOutputStream** використовує низку методів, серед яких можна виділити такі:

- **void close():** закриває потік;
- **void flush():** очищає буфер і скидає його вміст у вихідний потік;
- **void write(byte[] buf):** записує в потік масив байтів;
- **void write(int val):** записує в потік один молодший байт з val;
- **void writeBoolean(boolean val):** записує в потік значення Boolean;
- **void writeByte(int val):** записує в потік один молодший байт з val;
- **void writeChar(int val):** записує в потік значення типу char, представлене цілочисельним значенням;
- **void writeDouble(double val):** записує в потік значення типу double;
- **void writeFloat(float val):** записує в потік значення типу float;
- **void writeInt(int val):** записує цілочисельне значення int;
- **void writeLong(long val):** записує значення типу long;
- **void writeShort(int val):** записує значення типу short;

Серіалізація

- **void writeShort(int val):** записує значення типу short;
- **void writeUTF(String str):** записує в потік рядок у кодуванні UTF-8;
- **void writeObject(Object obj):** записує в потік окремий об'єкт.

Ці методи охоплюють весь спектр даних, які можна серіалізувати. Наприклад, збережемо у файл один об'єкт класу Person:

```
try(ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("person.dat"))){
    Person p = new Person("Sam", 33, 178, true);
    oos.writeObject(p);
}
catch(Exception ex){
    System.out.println(ex.getMessage());
}
```

```
class Person implements Serializable{
    2 usages
    private String name;
    2 usages
    private int age;
    2 usages
    private double height;
    2 usages
    private boolean married;
    1 usage
    Person(String n, int a, double h, boolean m){
        name=n;
        age=a;
        height=h;
        married=m;
    }
    no usages
    String getName() {return name;}
    no usages
    int getAge(){ return age;}
    no usages
    double getHeight(){return height;}
    no usages
    boolean getMarried(){return married;}
}
```

Серіалізація

Десеріалізація. Клас `ObjectInputStream`

Клас `ObjectInputStream` відповідає за зворотний процес - читання раніше серіалізованих даних із потоку. У конструкторі він приймає посилання на потік введення:

`ObjectInputStream(InputStream in)`

Функціонал `ObjectInputStream` зосереджений у методах, призначених для читання різних типів даних. Розглянемо основні методи цього класу:

- **`void close()`**: закриває потік;
- **`int skipBytes(int len)`**: пропускає при читанні кілька байт, кількість яких дорівнює `len`;
- **`int available()`**: повертає кількість байт, доступних для читання;
- **`int read()`**: зчитує з потоку один байт і повертає його цілочисельне представлення;
- **`boolean readBoolean()`**: зчитує з потоку одне значення `Boolean`;
- **`byte readByte()`**: зчитує з потоку один байт;
- **`char readChar()`**: зчитує з потоку один символ `char`;
- **`double readDouble()`**: зчитує значення типу `double`;
- **`float readFloat()`**: зчитує з потоку значення типу `float`;
- **`int readInt()`**: зчитує цілочисельне значення `int`;
- **`long readLong()`**: зчитує значення типу `long`;
- **`short readShort()`**: зчитує значення типу `short`;
- **`String readUTF()`**: зчитує рядок у кодуванні UTF-8;
- **`Object readObject()`**: зчитує з потоку об'єкт.

Серіалізація

Наприклад, витягнемо вище збережений об'єкт Person з файлу:

```
try(ObjectInputStream ois = new ObjectInputStream(new FileInputStream("person.dat"))) {
    Person p=(Person)ois.readObject();
    System.out.printf("Name: %s \t Age: %d \n", p.getName(), p.getAge());
}
catch(Exception ex){
    System.out.println(ex.getMessage());
}
```

Тепер поєднаємо збереження і відновлення з файлу на прикладі списку об'єктів:

```
String filename = "people.dat";
// створимо список об'єктів, які будемо записувати
ArrayList<Person> people = new ArrayList<Person>();
people.add(new Person("Tom", 30, 175, false));
people.add(new Person("Sam", 33, 178, true));

try(ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(filename))) {
    oos.writeObject(people);
    System.out.println("File has been written");
}
catch(Exception ex) {
    System.out.println(ex.getMessage());
}

// десеріалізація в новий список
ArrayList<Person> newPeople= new ArrayList<>();
try(ObjectInputStream ois = new ObjectInputStream(new FileInputStream(filename))) {
    newPeople=((ArrayList<Person>)ois.readObject());
}
catch(Exception ex){
    System.out.println(ex.getMessage());
}

for(Person p : newPeople)
    System.out.printf("Name: %s \t Age: %d \n", p.getName(), p.getAge());
```

Серіалізація

Виключення даних із серіалізації

За замовчуванням серіалізуються всі змінні об'єкта. Однак, можливо, ми хочемо, щоб деякі поля були виключені із серіалізації. Для цього вони мають бути оголошені з модифікатором `transient`. Наприклад, виключимо із серіалізації об'єкта `Person` змінні **height** і **married**:

```
class Person implements Serializable{
    2 usages
    private String name;
    2 usages
    private int age;
    2 usages
    private transient double height;
    2 usages
    private transient boolean married;
    2 usages
    Person(String n, int a, double h, boolean m){
        name=n;
        age=a;
        height=h;
        married=m;
    }
    1 usage
    String getName() {return name;}
    1 usage
    int getAge(){ return age;}
    no usages
    double getHeight(){return height;}
    no usages
    boolean getMarried(){return married;}
}
```

Клас File. Робота з файлами та каталогами

Клас **File**, визначений у пакеті **java.io**, не працює безпосередньо з потоками. Його завданням є управління інформацією про файли та каталоги. Хоча на рівні операційної системи файли і каталоги відрізняються, але в Java вони описуються одним класом **File**. Залежно від того, що повинен представляти об'єкт **File** - файл або каталог, ми можемо використовувати один із конструкторів для створення об'єкта:

```
File(String filePath)
File(String filePath, String filename)
File(File dir, String filename)
```

Наприклад:

```
File dir1 = new File("C://SomeDir");// створюємо об'єкти для файлів, які
знаходяться в каталозі
File file1 = new File("C://SomeDir", "Hello.txt");
File file2 = new File(dir1, "Hello2.txt");
```

Клас File. Робота з файлами та каталогами

Клас **File** має низку методів, які дають змогу керувати файлами та каталогами. Розглянемо деякі з них:

- **boolean createNewFile():** створює новий файл за шляхом, який передано в конструктор. У разі вдалого створення повертає true, інакше false;
- **boolean delete():** видаляє каталог або файл за шляхом, який передано в конструктор. У разі вдалого видалення повертає true;
- **boolean exists():** перевіряє, чи існує за вказаним у конструкторі шляхом файл або каталог. І якщо файл або каталог існує, то повертає true, інакше повертає false;
- **String getAbsolutePath():** повертає абсолютний шлях для шляху, переданого в конструктор об'єкта;
- **String getName():** повертає коротке ім'я файлу або каталогу;
- **String getParent():** повертає ім'я батьківського каталогу;
- **boolean isDirectory():** повертає значення true, якщо за вказаним шляхом розташовується каталог;
- **boolean isFile():** повертає значення true, якщо за вказаним шляхом розташований файл;
- **boolean isHidden():** повертає значення true, якщо каталог або файл є прихованими;
- **long length():** повертає розмір файлу в байтах;
- **long lastModified():** повертає час останньої зміни файлу або каталогу. Значення представляє кількість мілісекунд, що минули з початку епохи Unix;

Клас File. Робота з файлами та каталогами

- **String[] list():** повертає масив файлів і підкаталогів, які знаходяться в певному каталозі;
- **File[] listFiles():** повертає масив файлів і підкаталогів, які знаходяться в певному каталозі;
- **boolean mkdir():** створює новий каталог і при вдалому створенні повертає значення true; **boolean renameTo(File dest):** перейменовує файл або каталог.

Робота з каталогами

Якщо об'єкт **File** представляє каталог, то його метод `isDirectory()` повертає `true`. І тому ми можемо отримати його вміст - вкладені підкаталоги та файли за допомогою методів `list()` і `listFiles()`. Отримаємо всі підкаталоги та файли в певному каталозі:

```
public static void main(String[] args) {  
    // визначаємо об'єкт для каталогу  
    File dir = new File("D://");  
    // якщо об'єкт представляє каталог  
    if (dir.isDirectory())  
    {  
        // отримуємо всі вкладені об'єкти в каталозі  
        for (File item : Objects.requireNonNull(dir.listFiles())) {  
            if (item.isDirectory())  
                System.out.println(item.getName() + " \t folder");  
            else  
                System.out.println(item.getName() + " \t file");  
        }  
    }  
}
```


Клас File. Робота з файлами та каталогами

Тепер виконаємо ще низку операцій із каталогами, як-от видалення, перейменування та створення:

```
public static void main(String[] args) {  
    // визначаємо об'єкт для каталогу  
    File dir = new File( pathname: "D:/NewDir");  
    boolean created = dir.mkdir();  
    if (created)  
        System.out.println("Folder has been created");  
    // перейменуємо каталог  
    File newDir = new File( pathname: "D:/SomeDir/NewDirRenamed");  
    dir.renameTo(newDir);  
    // видалимо каталог  
    boolean deleted = newDir.delete();  
    if (deleted)  
        System.out.println("Folder has been deleted");  
}
```

Робота з ZIP-архівами

Крім загального функціоналу для роботи з файлами Java надає функціональність для роботи з таким видом файлів як zip-архіви. Для цього в пакеті **java.util.zip** визначено два класи - **ZipInputStream** і **ZipOutputStream**

ZipOutputStream. Запис архівів

Для створення архіву використовується клас **ZipOutputStream**. Для створення об'єкта **ZipOutputStream** у його конструктор передається потік виведення: **ZipOutputStream(OutputStream out)**

Для запису файлів в архів для кожного файлу створюється об'єкт **ZipEntry**, у конструктор якого передається ім'я архівованого файлу. А щоб додати кожен об'єкт **ZipEntry** в архів, застосовується метод **putNextEntry()**. Створимо архів:

```
String filename = "notes.txt";
try (ZipOutputStream zout = new ZipOutputStream(new FileOutputStream("output.zip"));
    FileInputStream fis = new FileInputStream(filename)) {
    ZipEntry entry1 = new ZipEntry("notes.txt");
    zout.putNextEntry(entry1);
    // читаємо вміст файлу в масив byte
    byte[] buffer = new byte[fis.available()];
    fis.read(buffer);
    // додаємо вміст до архіву
    zout.write(buffer);
    // закриваємо поточний запис для нового запису
    zout.closeEntry();
}
catch (Exception ex) {
    System.out.println(ex.getMessage());
}
```

Робота з ZIP-архівами

Після додавання об'єкта **ZipEntry** в потік нам також треба додати в нього і вміст файлу. Для цього використовується метод `write`, що записує в потік масив байтів: **`zout.write(buffer);`**. Наприкінці треба закрити **ZipEntry** за допомогою методу **`closeEntry()`**. Після цього можна додавати в архів нові файли - у цьому випадку всі вищеописані дії для кожного нового файлу повторюються.

Читання архівів. **ZipInputStream**

Для читання архівів застосовується клас **ZipInputStream**. У конструкторі він приймає потік, що вказує на zip-архів: **`ZipInputStream(InputStream in)`**

Для зчитування файлів з архіву **ZipInputStream** використовує метод **`getNextEntry()`**, який повертає об'єкт **ZipEntry**. Об'єкт **ZipEntry** представляє окремий запис у zip-архіві. Наприклад, зчитаємо який-небудь архів:

```
public static void main(String[] args) {
    try(ZipInputStream zin = new ZipInputStream(new FileInputStream("output.zip"))) {
        ZipEntry entry;
        String name;
        while((entry=zin.getNextEntry())!=null){
            name = entry.getName(); // отримаємо назву файлу
            System.out.printf("File name: %s \n", name);
            // розпакування
            FileOutputStream fout = new FileOutputStream("new" + name);
            for (int c = zin.read(); c != -1; c = zin.read())
                fout.write(c);

            fout.flush();
            zin.closeEntry();
            fout.close();
        }
    }
    catch(Exception ex){
        System.out.println(ex.getMessage());
    }
}
```

Робота з ZIP-архівами

ZipInputStream у конструкторі отримує посилання на потік введення. І потім у циклі виводяться всі файли та їхній розмір у байтах, які знаходяться в цьому архіві. Потім дані витягуються з архіву і зберігаються в нові файли, які знаходяться в тій же папці і які починаються з "new".

Контрольні питання

1. Вкажіть класи, які є вершинами символічної ієрархії потоків введення-виводу в мові програмування Java.
2. Визначте класи, які є вершинами байтової ієрархії потоків введення-виводу в мові програмування Java.
3. Як визначається термін "потік" при взаємодії з файловою системою в контексті мови програмування Java?
4. Охарактеризуйте різницю між класами `OutputStream`, `InputStream`, `Writer` та `Reader` у мові програмування Java.
5. Які методи класу `File` вам відомі в мові програмування Java?
6. Для чого використовуються класи `BufferedReader` та `BufferedWriter` у мові програмування Java?

The END
Lec 10.

.NET та JAVA #11

Введення в Stream API та лямбда-вирази

1. Лямбда-вирази
2. Вбудовані функціональні інтерфейси
3. Stream API
4. Тип Optional
5. Паралельні потоки

Лямбда-вирази

Серед нововведень, які були привнесені в мову Java з виходом **JDK 8**, окремо стоять **лямбда-вирази**. **Лямбда** являє собою набір інструкцій, які можна виділити в окрему змінну і потім багаторазово викликати в різних місцях програми.

Основу лямбда-виразу становить лямбда-оператор, який представляє стрілку **->**. Цей оператор розділяє лямбда-вираз на дві частини: ліва частина містить список параметрів виразу, а права власне представляє тіло лямбда-виразу, де виконуються всі дії.

Лямбда-вираз не виконується сам по собі, а утворює реалізацію методу, визначеного у функціональному інтерфейсі. При цьому важливо, що функціональний інтерфейс повинен містити тільки один єдиний метод без реалізації.

```
interface Operationable{
    1 usage
    int calculate(int x, int y);
}

class Program {
    public static void main(String[] args) {
        Operationable operation;
        operation = (x,y)->x+y;

        int result = operation.calculate( x: 10, y: 20);
        System.out.println(result); //30
    }
}
```


Лямбда-вирази

У ролі функціонального інтерфейсу виступає інтерфейс **Operationable**, у якому визначено один метод без реалізації - метод `calculate`. Цей метод приймає два параметри - цілих числа, і повертає деяке ціле число.

За фактом **лямбда-вирази** є до певної міри скороченою формою внутрішніх анонімних класів, які раніше застосовувалися в **Java**. Зокрема, попередній приклад ми можемо переписати таким чином:

```
Operationable op = new Operationable(){
    1 usage
    public int calculate(int x, int y){
        return x + y;
    }
};
int z = op.calculate( x: 20, y: 10);
System.out.println(z); // 30
```

Щоб оголосити і використовувати лямбда-вираз, основна програма розбивається на низку етапів:

- 1) Визначення посилання на функціональний інтерфейс: `Operationable operation;`
- 2) Створення лямбда-виразу: `operation = (x,y) -> x+y;` Причому параметри лямбда-виразу відповідають параметрам єдиного методу інтерфейсу `Operationable`, а результат відповідає результату методу інтерфейсу, що повертається. При цьому нам не треба використовувати ключове слово `return` для повернення результату з лямбда-виразу.

Лямбда-вирази

Так, у методі інтерфейсу обидва параметри представляють тип `int`, отже, у тілі лямбда-виразу ми можемо застосувати до них додавання. Результат додавання також представляє тип `int`, об'єкт якого повертається методом інтерфейсу.

3) Використання лямбда-виразу у вигляді виклику методу інтерфейсу:

```
int result = operation.calculate(10, 20);
```

Оскільки в лямбда-виразі визначено операцію додавання параметрів, результатом методу буде сума чисел 10 і 20.

При цьому для одного функціонального інтерфейсу ми можемо визначити безліч лямбда-виразів. Наприклад:

```
Operationable operation1 = (int x, int y)-> x + y;  
Operationable operation2 = (int x, int y)-> x - y;  
Operationable operation3 = (int x, int y)-> x * y;  
  
System.out.println(operation1.calculate(x: 20, y: 10)); //30  
System.out.println(operation2.calculate(x: 20, y: 10)); //10  
System.out.println(operation3.calculate(x: 20, y: 10)); //200
```

Лямбда-вирази

Відкладене виконання

Одним із ключових моментів у використанні лямбд є відкладене виконання (**deferred execution**). Тобто ми визначаємо в одному місці програми лямбда-вираз і потім можемо його викликати за необхідності невизначену кількість разів у різних частинах програми. Відкладене виконання може знадобитися, наприклад, у таких випадках:

- Виконання коду в окремому потоці;
- Виконання одного і того ж коду кілька разів;
- Виконання коду в результаті якоїсь події;
- Виконання коду тільки в тому випадку, коли він дійсно необхідний і якщо він необхідний.

Передання параметрів у лямбда-вираз

Параметри лямбда-виразу повинні відповідати за типом параметрам методу з функціонального інтерфейсу. Під час написання самого лямбда-виразу тип параметрів писати необов'язково, хоча в принципі це можна зробити, наприклад:

```
operation = (int x, int y)->x+y;
```

Якщо метод не приймає жодних параметрів, то пишуться порожні дужки, наприклад:

```
()-> 30 + 20;
```

Якщо метод приймає тільки один параметр, то дужки можна опустити:

```
n-> n * n;
```

Лямбда-вирази

Термінальні лямбда-вирази

До цього ми розглянули **лямбда-вирази**, які повертають певне значення. Але також можуть бути й термінальні лямбди, які не повертають жодного значення. Наприклад:

```
interface Printable{
    1 usage
    void print(String s);
}

class Program {
    public static void main(String[] args) {
        Printable printer = s->System.out.println(s);
        printer.print("Hello Java!");
    }
}
```

Лямбда-вирази

Лямбди та локальні змінні

Лямбда-вираз може використовувати змінні, які оголошені зовні в більш загальній області видимості - на рівні класу або методу, в якому лямбда-вираз визначено. Однак залежно від того, як і де визначено змінні, можуть різнитися способи їхнього використання в лямбдах. Розглянемо перший приклад - використання змінних рівня класу:

```
interface Operation{
    1 usage
    int calculate();
}

class Program {
    3 usages
    static int x = 10;
    1 usage
    static int y = 20;
    public static void main(String[] args) {
        Operation op = ()->{
            x=30;
            return x+y;
        };
        System.out.println(op.calculate()); // 50
        System.out.println(x); // 30 - значення x змінилось
    }
}
```

Змінні **x** і **y** оголошені на рівні класу, і в лямбда-виразі ми їх можемо отримати і навіть змінити. Так, у цьому випадку після виконання виразу змінюється значення змінної **x**.

Лямбда-вирази

Тепер розглянемо інший приклад - локальні змінні на рівні методу:

```
int n=70;
int m=30;
Operation op = ()->{
    //n=100; - так не можна зробити
    return m+n;
};
// n=100; - так також не можна зробити
System.out.println(op.calculate()); // 100
```

Локальні змінні рівня методу ми також можемо використовувати в лямбдах, але змінювати їхнє значення не можна. Якщо ми спробуємо це зробити, то середовище розробки може нам висвітити помилку і те, що таку змінну треба позначити за допомогою ключового слова `final`, тобто зробити константою:

```
final int n=70;.
```

Однак це необов'язково. Ба більше, ми не зможемо змінити значення змінної, яка використовується в лямбда-виразі, поза цим виразом. Тобто навіть якщо така змінна не оголошена як константа, по суті вона є константою.

Лямбда-вирази

Блоки коду в лямбда-виразах

Існують два типи лямбда-виразів: однорядковий вираз і блок коду. Приклади однорядкових виразів демонструвалися вище. Блокові вирази обрамляються фігурними дужками. У блокових лямбда-виразах можна використовувати внутрішні вкладені блоки, цикли, конструкції **if**, **switch**, створювати змінні тощо. Якщо блоковий лямбда-вираз має повертати значення, то явним чином застосовується оператор **return**:

```
interface Operationable{
    2 usages
    int calculate(int x, int y);
}

class Program {
    public static void main(String[] args) {
        Operationable operation = (int x, int y)-> {
            if(y==0)
                return 0;
            else
                return x/y;
        };
        System.out.println(operation.calculate(x: 20, y: 10)); //2
        System.out.println(operation.calculate(x: 20, y: 0)); //0
    }
}
```

Лямбда-вирази

Узагальнений функціональний інтерфейс

Функціональний інтерфейс може бути узагальненим, однак у лямбда-виразі використання узагальнень не допускається. У цьому випадку нам треба типізувати об'єкт інтерфейсу певним типом, який потім буде застосовуватися в лямбда-виразі. Приклад приведено нижче, таким чином, під час оголошення лямбда-виразу йому вже відомо, який тип параметрів будуть представляти і який тип вони будуть повертати.

```
interface Operationable<T>{  
    2 usages  
    T calculate(T x, T y);  
}  
  
class Program {  
    public static void main(String[] args) {  
        Operationable<Integer> operation1 = (x, y)-> x + y;  
        Operationable<String> operation2 = (x, y) -> x + y;  
        System.out.println(operation1.calculate(x: 20, y: 10)); //30  
        System.out.println(operation2.calculate(x: "20", y: "10")); //2010  
    }  
}
```


Лямбда-вирази

Лямбди як параметри методів

Однією з переваг лямбд у java є те, що їх можна передавати як параметри в методи. Розглянемо приклад:

```
interface Expression{
    1 usage
    boolean isEqual(int n);
}

class LambdaApp {
    public static void main(String[] args) {
        Expression func = (n)-> n%2==0;
        int[] nums = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
        System.out.println(sum(nums, func)); // 20
    }
    1 usage
    private static int sum (int[] numbers, Expression func){
        int result = 0;
        for(int i : numbers)
            if (func.isEqual(i))
                result += i;

        return result;
    }
}
```

Функціональний інтерфейс **Expression** визначає метод **isEqual()**, який повертає true, якщо стосовно числа n діє якась рівність. В основному класі програми визначається метод **sum()**, який обчислює суму всіх елементів масиву, що відповідають деякій умові. А сама умова передається через параметр **Expression func**. Причому на момент написання методу **sum** ми можемо абсолютно не знати, яку саме умову буде використано.

Лямбда-вирази

Посилання на метод як параметри методів

Починаючи з **JDK 8** у Java можна як параметр у метод передавати посилання на інший метод. У принципі цей спосіб аналогічний передачі в метод лямбда-виразу. Посилання на метод передається у вигляді **ім'я_класу::ім'я_статичного_методу** (якщо метод статичний) або **об'єкт_класу::ім'я_методу** (якщо метод нестатичний). Розглянемо на прикладі:

```
// функціональний інтерфейс
2 usages
interface Expression{
    1 usage
    boolean isEqual(int n);
}
// клас, в якому визначені методи
2 usages
class ExpressionHelper{
    1 usage
    static boolean isEven(int n){
        return n%2 == 0;
    }
    1 usage
    static boolean isPositive(int n){
        return n > 0;
    }
}
```

```
class Program {
    public static void main(String[] args) {
        int[] nums = { -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5};
        System.out.println(sum(nums, ExpressionHelper::isEven));

        Expression expr = ExpressionHelper::isPositive;
        System.out.println(sum(nums, expr));
    }

    2 usages
    private static int sum (int[] numbers, Expression func) {
        int result = 0;
        for(int i : numbers)
            if (func.isEqual(i))
                result += i;

        return result;
    }
}
```

Лямбда-вирази

Тут також визначено функціональний інтерфейс **Expression**, який має один метод. Крім того, визначено клас **ExpressionHelper**, який містить два статичні методи. У принципі їх можна було визначити і в основному класі програми, але я виніс їх в окремий клас. В основному класі програми LambdaApp визначено метод `sum()`, який повертає суму елементів масиву, що відповідають деякій умові. Умова передається у вигляді об'єкта функціонального інтерфейсу `Expression`. У методі `main` двічі викликаємо метод `sum`, передаючи в нього один і той самий масив чисел, але різні умови. Перший виклик методу `sum`:

```
System.out.println(sum(nums, ExpressionHelper::isEven));
```

На місце другого параметра передається `ExpressionHelper::isEven`, тобто посилання на статичний метод `isEven()` класу `ExpressionHelper`.

При цьому методи, на які йде посилання, повинні збігатися за параметрами і результатом з методом функціонального інтерфейсу. При другому виклику методу `sum` окремо створюється об'єкт `Expression`, який потім передається в метод:

```
Expression expr = ExpressionHelper::isPositive;  
System.out.println(sum(nums, expr));
```

Використання посилань на методи як параметрів аналогічно використанню лямбда-виразів. Якщо нам треба викликати нестатичні методи, то в посиланні замість імені класу застосовується ім'я об'єкта цього класу:

Лямбда-вирази

```
interface Expression{  
    1usage  
    boolean isEqual(int n);  
}  
  
2 usages  
class ExpressionHelper{  
    1usage  
    boolean isEven(int n){  
        return n%2 == 0;  
    }  
}
```

```
class Program {  
    public static void main(String[] args) {  
        int[] nums = { -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5};  
        ExpressionHelper exprHelper = new ExpressionHelper();  
        System.out.println(sum(nums, exprHelper::isEven)); // 0  
    }  
  
    1usage  
    private static int sum (int[] numbers, Expression func) {  
        int result = 0;  
        for(int i : numbers)  
            if (func.isEqual(i))  
                result += i;  
  
        return result;  
    }  
}
```

Лямбда-вирази

Посилання на конструктори

Подібним чином ми можемо використовувати конструктори: `назва_класу::new`.
Наприклад:

```
interface UserBuilder{
    1 usage
    User create(String name);
}

3 usages
class User{
    2 usages
    private String name;
    1 usage
    String getName(){
        return name;
    }
    1 usage
    User(String n){
        this.name=n;
    }
}
```

```
class Program {
    public static void main(String[] args) {
        UserBuilder userBuilder = User::new;
        User user = userBuilder.create("Tom");
        System.out.println(user.getName());
    }
}
```

У разі використання конструкторів методи функціональних інтерфейсів повинні приймати той самий список параметрів, що й конструктори класу, і повинні повертати об'єкт цього класу.

Лямбда-вирази

Лямбди як результат методів

Також метод у Java може повертати лямбда-вираз. Розглянемо наступний приклад:

```
interface Operation{
    2 usages
    int execute(int x, int y);
}

class Program {
    public static void main(String[] args) {
        Operation func = action( number: 1);
        int a = func.execute( x: 6, y: 5);
        System.out.println(a);           // 11
        int b = action( number: 2).execute( x: 8, y: 2);
        System.out.println(b);           // 6
    }

    2 usages
    private static Operation action(int number){
        return switch (number) {
            case 1 -> (x, y) -> x + y;
            case 2 -> (x, y) -> x - y;
            case 3 -> (x, y) -> x * y;
            default -> (x, y) -> 0;
        };
    }
}
```

У цьому випадку визначено функціональний інтерфейс **Operation**, у якому метод `execute` приймає два значення типу `int` і повертає значення типу `int`. Метод **action** приймає як параметр число і залежно від його значення повертає той чи інший лямбда-вираз. Він може представляти або додавання, або віднімання, або множення, або просто повертає 0. Варто враховувати, що формально типом методу `action`, який повертається, є інтерфейс `Operation`, а лямбда-вираз, який повертається, має відповідати цьому інтерфейсу. У методі `main` ми можемо викликати цей метод `action`. Наприклад, спочатку отримати його результат - лямбда-вираз, який присвоюється змінній `Operation`.

Вбудовані функціональні інтерфейси

У **JDK 8** разом із самою функціональністю лямбда-виразів також було додано деяку кількість вбудованих функціональних інтерфейсів, які ми можемо використовувати в різних ситуаціях і в різні API в рамках **JDK 8**. Зокрема, низка інтерфейсів, що розглядаються далі, широко застосовується в **Stream API** - новому прикладному інтерфейсі для роботи з даними. Розглянемо основні з цих інтерфейсів:

- **Predicate<T>**
- **Consumer<T>**
- **Function<T,R>**
- **Supplier<T>**
- **UnaryOperator<T>**
- **BinaryOperator<T>**

Predicate<T>

Функціональний інтерфейс **Predicate<T>** перевіряє дотримання деякої умови. Якщо вона дотримується, то повертається значення **true**. Як параметр лямбда-вираз приймає об'єкт типу **T**:

```
public interface Predicate<T> {boolean test(T t);}
```

```
class Program {  
    public static void main(String[] args) {  
        BinaryOperator<Integer> multiply = (x, y) -> x*y;  
  
        System.out.println(multiply.apply( 3, 5)); // 15  
        System.out.println(multiply.apply( 10, -2)); // -20  
    }  
}
```

Вбудовані функціональні інтерфейси

BinaryOperator<T>

BinaryOperator<T> приймає як параметр два об'єкти типу T, виконує над ними бінарну операцію і повертає її результат також у вигляді об'єкта типу T:

```
public interface BinaryOperator<T> {T apply(T t1, T t2);}
```

```
class Program {
    public static void main(String[] args) {
        BinaryOperator<Integer> multiply = (x, y) -> x*y;

        System.out.println(multiply.apply(3, 5)); // 15
        System.out.println(multiply.apply(10, -2)); // -20
    }
}
```

UnaryOperator<T>

UnaryOperator<T> приймає як параметр об'єкт типу T, виконує над ними операції і повертає результат операцій у вигляді об'єкта типу T:

```
public interface UnaryOperator<T> {T apply(T t);}
```

```
class Program {
    public static void main(String[] args) {
        UnaryOperator<Integer> square = x -> x*x;
        System.out.println(square.apply(5)); // 25
    }
}
```

Function<T,R>

Функціональний інтерфейс Function<T,R> представляє функцію переходу від об'єкта типу T до об'єкта типу R:

```
public interface Function<T, R> {R apply(T t);}
```

```
class Program {
    public static void main(String[] args) {
        Function<Integer, String> convert = x -> String.valueOf(x) + " dollars";
        System.out.println(convert.apply(5)); // 5 dollars
    }
}
```


Вбудовані функціональні інтерфейси

Consumer<T>

Consumer<T> виконує деяку дію над об'єктом типу T, при цьому нічого не повертаючи:

```
public interface Consumer<T> {  
    void accept(T t);  
}
```

```
class Program {  
    public static void main(String[] args) {  
        Consumer<Integer> printer = x-> System.out.printf("%d dollars \n", x);  
        printer.accept(600); // 600 dollars  
    }  
}
```

Supplier<T>

Supplier<T> не приймає жодних аргументів, але повинен повертати об'єкт типу T:

```
public interface Supplier<T> {  
    T get();  
}
```

```
class Program{  
    public static void main(String[] args) {  
        Supplier<User> userFactory = ()->{  
            Scanner in = new Scanner(System.in);  
            System.out.println("Enter name: ");  
            String name = in.nextLine();  
            return new User(name);  
        };  
        User user1 = userFactory.get();  
        User user2 = userFactory.get();  
        System.out.println("Name of user1: " + user1.getName());  
        System.out.println("Name of user2: " + user2.getName());  
    }  
}  
4 usages  
class User{  
    2 usages  
    private String name;  
    2 usages  
    String getName(){return name;}  
    1 usage  
    User(String n){this.name=n;}  
}
```

Stream API

Починаючи з **JDK 8** у Java з'явився **Stream API**. Його завдання - спростити роботу з наборами даних, зокрема, спростити операції фільтрації, сортування та інші маніпуляції з даними. Уся основна функціональність цього API зосереджена в пакеті **java.util.stream**. Ключовим поняттям у **Stream API** є потік даних. Як вже було зазначено в минулій лекції, термін "**потік**" є доволі перевантаженим у програмуванні загалом і в Java зокрема. Стосовно Stream API потік являє собою канал передачі даних із джерела даних. Причому в якості джерела можуть виступати як файли, так і масиви та колекції.

Однією з відмінних рис **Stream API** є застосування лямбда-виразів, які дозволяють значно скоротити запис виконуваних дій. При найближчому розгляді ми можемо знайти в інших технологіях програмування аналоги подібного API. Зокрема, у мові **C#** деяким аналогом **Stream API** буде технологія **LINQ**.

Розглянемо найпростіший приклад. Припустімо, у нас є завдання: знайти в масиві кількість усіх чисел, які більші за 0. До JDK 8 ми б могли написати лічильник таким чином:

```
int[] numbers = {-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5};
int count=0;
for(int i:numbers)
    if(i > 0) count++;

System.out.println(count);
```

Stream API

Тепер застосуємо Stream API:

```
long count = IntStream.of( ...values: -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5).filter(w -> w > 0).count();  
System.out.println(count);
```

Тепер замість циклу і купи умовних конструкцій, які ми б використовували до **JDK 8**, ми можемо записати ланцюжок методів, які будуть виконувати ті ж дії.

Під час роботи зі Stream API важливо розуміти, що всі операції з потоками бувають або **термінальними** (terminal), або **проміжними** (intermediate).

Проміжні операції повертають трансформований потік. Наприклад, вище в прикладі метод filter приймав потік чисел і повертав уже перетворений потік, у якому тільки числа більші за 0. До повернутого потоку також можна застосувати низку проміжних операцій. Кінцеві або термінальні операції повертають конкретний результат. Наприклад, у прикладі вище метод count() представляє термінальну операцію і повертає число. Після цього ніяких проміжних операцій природно застосовувати не можна.

Усі потоки виконують обчислення, зокрема в проміжних операціях, тільки тоді, коли до них застосовується термінальна операція. Тобто в цьому випадку застосовується відкладене виконання.

Stream API

В основі Stream API лежить інтерфейс **BaseStream**. Його повне визначення:

```
interface BaseStream<T , S extends BaseStream<T , S>>
```

Тут параметр T означає тип даних у потоці, а S - тип потоку, який успадковується від інтерфейсу **BaseStream**. Базовий функціонал реалізується через методи:

- **void close()**: закриває потік;
- **boolean isParallel()**: повертає true, якщо потік є паралельним;
- **Iterator<T> iterator()**: повертає посилання на ітератор потоку;
- **Splitter<T> splitter()**: повертає посилання на сплітератор потоку;
- **S parallel()**: повертає паралельний потік (паралельні потоки можуть задіяти кілька ядер процесора в багатоядерних архітектурах);
- **S sequential()**: повертає послідовний потік;
- **S unordered()**: повертає невпорядкований потік.

Від інтерфейсу BaseStream успадковується низка інтерфейсів, призначених для створення конкретних потоків:

- **Stream<T>**: використовується для потоків даних, що представляють будь-який посилальний тип;
- **IntStream**: використовується для потоків із типом даних int;
- **DoubleStream**: використовується для потоків із типом даних double;
- **LongStream**: використовується для потоків з типом даних long.

При роботі з потоками, які представляють певний примітивний тип - **double, int, long** простіше використовувати інтерфейси **DoubleStream, IntStream, LongStream**. Але в більшості випадків, як правило, робота відбувається зі складнішими даними, для яких призначений інтерфейс **Stream<T>**. Розглянемо деякі його методи на наступному слайді.

Stream API

- **boolean allMatch(Predicate<? super T> predicate):** повертає true, якщо всі елементи потоку задовольняють умові в предикаті. Термінальна операція;
- **boolean anyMatch(Predicate<? super T> predicate):** повертає true, якщо хоч один елемент потоку задовольняє умову в предикаті. Термінальна операція;
- **<R,A> R collect(Collector<? super T,A,R> collector):** додає елементи до незмінного контейнера з типом R. T представляє тип даних із потоку, що викликає, а A - тип даних у контейнері. Термінальна операція;
- **long count():** повертає кількість елементів у потоці. Термінальна операція;
- **Stream<T> concat(Stream<? extends T> a, Stream<? extends T> b):** об'єднує два потоки. Проміжна операція;
- **Stream<T> distinct():** повертає потік, у якому є тільки унікальні дані з типом T. Проміжна операція;
- **Stream<T> dropWhile(Predicate<? super T> predicate):** пропускає елементи, які відповідають умові в predicate, доки не трапиться елемент, який не відповідає умові. Обрані елементи повертаються у вигляді потоку. Проміжна операція;
- **Stream<T> filter(Predicate<? super T> predicate):** фільтрує елементи відповідно до умови в предикаті. Проміжна операція;
- **Optional<T> findFirst():** повертає перший елемент з потоку. Термінальна операція;
- **Optional<T> findAny():** повертає перший-ліпший елемент із потоку. Термінальна операція;

Stream API

- **void forEach(Consumer<? super T> action):** для кожного елемента виконується дія action. Термінальна операція;
- **Stream<T> limit(long maxSize):** залишає в потоці тільки maxSize елементів. Проміжна операція;
- **Optional<T> max(Comparator<? super T> comparator):** повертає максимальний елемент із потоку. Для порівняння елементів застосовується компаратор comparator. Термінальна операція;
- **Optional<T> min(Comparator<? super T> comparator):** повертає мінімальний елемент з потоку. Для порівняння елементів застосовується компаратор comparator. Термінальна операція;
- **<R> Stream<R> map(Function<? super T,? extends R> mapper):** перетворює елементи типу T на елементи типу R і повертає потік з елементами R. Проміжна операція;
- **<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper):** дає змогу перетворити елемент типу T на кілька елементів типу R і повертає потік з елементами R. Проміжна операція;

Stream API

- **boolean noneMatch(Predicate<? super T> predicate):** повертає true, якщо жоден з елементів у потоці не задовольняє умові в предикаті. Термінальна операція;
- **Stream<T> skip(long n):** повертає потік, у якому відсутні перші n елементів. Проміжна операція;
- **Stream<T> sorted():** повертає відсортований потік. Проміжна операція;
- **Stream<T> sorted(Comparator<? super T> comparator):** повертає відсортований відповідно до компаратора потік. Проміжна операція;
- **Stream<T> takeWhile(Predicate<? super T> predicate):** вибирає з потоку елементи, поки вони відповідають умові в predicate. Обрані елементи повертаються у вигляді потоку. Проміжна операція;
- **Object[] toArray():** повертає масив з елементів потоку. Термінальна операція.

Stream API

Незважаючи на те, що всі ці операції дають змогу взаємодіяти з потоком як певним набором даних на кшталт колекції, важливо розуміти відмінність колекцій від потоків:

- **Потоки не зберігають елементи.** Елементи, що використовуються в потоках, можуть зберігатися в колекції, або за необхідності можуть бути безпосередньо згенеровані;
- **Операції з потоками не змінюють джерела даних.** Операції з потоками лише повертають новий потік із результатами цих операцій;
- **Для потоків характерне відкладене виконання.** Тобто виконання всіх операцій з потоком відбувається лише тоді, коли виконується термінальна операція і повертається конкретний результат, а не новий потік.

Створення потоку даних

Для створення потоку даних можна застосовувати різні методи. Як джерело потоку ми можемо використовувати колекції. Зокрема, в JDK 8 в інтерфейс Collection, який реалізується всіма класами колекцій, було додано два методи для роботи з потоками:

- `default Stream<E> stream`: повертається потік даних із колекції;
- `default Stream<E> parallelStream`: повертається паралельний потік даних з колекції

Так, розглянемо приклад з ArrayList.

```
ArrayList<String> cities = new ArrayList<>();  
Collections.addAll(cities, "Київ", "Луцьк", "Львів", "Дніпро");  
cities.stream() // отримуємо потік  
    .filter(s->s.length()==6) // застосовуємо фільтрацію за довжиною рядка  
    .forEach(s->System.out.println(s)); // виводимо відфільтровані рядки на консоль
```


Stream API

Тут за допомогою виклику **cities.stream()** отримуємо потік, який використовує дані зі списку **cities**. За допомогою кожної проміжної операції, яка застосовується до потоку, ми також можемо отримати потік з урахуванням модифікацій. Наприклад, ми можемо змінити попередній приклад таким чином:

```
ArrayList<String> cities = new ArrayList<>();
Collections.addAll(cities, "Донецьк", "Харків", "Луганськ");

Stream<String> citiesStream = cities.stream(); // отримуємо потік
citiesStream = citiesStream.filter(s->s.length()==6); // застосовуємо фільтрацію за довжиною рядка
citiesStream.forEach(s->System.out.println(s)); // виводимо відфільтровані рядки на консоль
```

Важливо, що після використання термінальних операцій інші термінальні або проміжні операції до цього ж потоку не можуть бути застосовані, потік уже спожито.

```
ArrayList<String> cities = new ArrayList<>();
Collections.addAll(cities, "Донецьк", "Харків", "Луганськ");

Stream<String> citiesStream = cities.stream(); // отримуємо потік
citiesStream.forEach(s->System.out.println(s)); // термінальна операція вживає потік
long number = citiesStream.count(); // тут помилка, оскільки потік уже вжито
System.out.println(number);
citiesStream = citiesStream.filter(s->s.length()>5); // теж не можна, оскільки потік уже спожито
```

Stream API

Фактично життєвий цикл потоку проходить такі **три стадії**:

- Створення потоку;
- Застосування до потоку низки проміжних операцій;
- Застосування до потоку термінальної операції та отримання результату;

Крім вищерозглянутих методів ми можемо використовувати ще низку способів для створення потоку даних. Один з таких способів представляє метод **`Arrays.stream(T[] array)`**, який створює потік даних з масиву:

```
Stream<String> citiesStream = Arrays.stream(new String[]{"Донецьк", "Харків", "Луганськ"});  
citiesStream.forEach(s->System.out.println(s)); // виводимо всі елементи масиву
```

Для створення потоків **`IntStream`**, **`DoubleStream`**, **`LongStream`** можна використовувати відповідні перевантажені версії цього методу:

```
IntStream intStream = Arrays.stream(new int[]{1,2,4,5,7});  
intStream.forEach(i->System.out.println(i));  
  
LongStream longStream = Arrays.stream(new long[]{100,250,400,5843787,237});  
longStream.forEach(l->System.out.println(l));  
  
DoubleStream doubleStream = Arrays.stream(new double[] {3.4, 6.7, 9.5, 8.2345, 121});  
doubleStream.forEach(d->System.out.println(d));
```

Stream API

І ще один спосіб створення потоку представляє статичний метод `of(T..values)` класу `Stream`:

```
String[] cities = {"Донецьк", "Харків", "Луганськ"};
Stream<String> citiesStream2 = Stream.of(cities);

IntStream intStream = IntStream.of(...values: 1,2,4,5,7);
intStream.forEach(i->System.out.println(i));

LongStream longStream = LongStream.of(...values: 100,250,400,5843787,237);
longStream.forEach(l->System.out.println(l));

DoubleStream doubleStream = DoubleStream.of(...values: 3.4, 6.7, 9.5, 8.2345, 121);
doubleStream.forEach(d->System.out.println(d));
```

Фільтрація, перебір елементів і відображення

Перебір елементів. Метод `forEach`

Для перебору елементів потоку застосовується метод `forEach()`, який представляє термінальну операцію. Як параметр він приймає об'єкт `Consumer<? super String>`, який представляє дію, що виконується для кожного елемента набору. Наприклад:

```
Stream<String> citiesStream = Stream.of(...values: "Донецьк", "Харків", "Луганськ");
citiesStream.forEach(s->System.out.println(s));
```

Фактично це буде аналогічно перебору всіх елементів у циклі `for` і виконанню з ними дії, а саме виведення на консоль. Альтернатива:

```
Stream<String> citiesStream = Stream.of(...values: "Донецьк", "Харків", "Луганськ");
citiesStream.forEach(System.out::println);
```

Stream API

Фільтрація. Метод `filter`

Для фільтрації елементів у потоці застосовується метод **`filter()`**, який представляє проміжну операцію. Він приймає як параметр деяку умову у вигляді об'єкта **`Predicate<T>`** і повертає новий потік з елементів, які задовольняють цій умові:

```
Stream<String> citiesStream = Stream.of( ...values: "Донецьк", "Харків", "Луганськ");  
citiesStream.filter(s->s.length()==6).forEach(System.out::println);
```

Тут умова `s.length()==6` повертає `true` для тих елементів, довжина яких дорівнює 6 символам.

Розглянемо ще один приклад фільтрації зі складнішими даними.

Stream API

Припустимо, у нас є такий клас Phone, відфільтруємо набір телефонів за ціною:

```
class Phone{  
    3 usages  
    private String name;  
    3 usages  
    private int price;  
  
    no usages  
    public Phone(String name, int price){  
        this.name=name;  
        this.price=price;  
    }  
    no usages  
    public String getName() {  
        return name;  
    }  
    no usages  
    public void setName(String name) {  
        this.name = name;  
    }  
    no usages  
    public int getPrice() {  
        return price;  
    }  
    no usages  
    public void setPrice(int price) {  
        this.price = price;  
    }  
}
```

```
Stream<Phone> phoneStream = Stream.of(new Phone( name: "iPhone 6 S", price: 54000), new Phone( name: "Lumia 950", price: 45000),  
    new Phone( name: "Samsung Galaxy S 6", price: 40000));  
  
phoneStream.filter(p->p.getPrice()<50000).forEach(p->System.out.println(p.getName()));
```

Відображення. Метод map

Відображення або маппінг дає змогу задати функцію перетворення одного об'єкта в інший, тобто отримати з елемента одного типу елемент іншого типу. Для відображення використовується метод map, який має таке визначення:

`<R> Stream<R> map(Function<? super T, ? extends R> mapper)`

Передана в метод map функція задає перетворення від об'єктів типу T до типу R. І в результаті повертається новий потік із перетвореними об'єктами.

Stream API

Візьмемо вищевизначений клас телефонів і виконаємо перетворення від типу **Phone** до типу **String**. Операція **map(p-> p.getName())** поміщає в новий потік тільки назви телефонів.

```
Stream<Phone> phoneStream = Stream.of(new Phone( name: "iPhone 6 S", price: 54000), new Phone( name: "Lumia 950", price: 45000),
    new Phone( name: "Samsung Galaxy S 6", price: 40000));

phoneStream
    .map(Phone::getName) // поміщаємо в потік тільки назви телефонів
    .forEach(System.out::println);
```

Плоске відображення. Метод flatMap

Плоске відображення виконується тоді, коли з одного елемента потрібно отримати кілька. Цю операцію виконує метод flatMap:

```
<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>>
mapper)
```

Наприклад, у прикладі вище ми виводимо назву телефону і його ціну. Але що, якщо ми хочемо встановити для кожного телефону ціну зі знижкою і ціну без знижки. Тобто з одного об'єкта Phone нам треба отримати два об'єкти з інформацією, наприклад, у вигляді рядка. Для цього застосуємо flatMap.

Stream API

```
Stream<Phone> phoneStream = Stream.of(new Phone( name: "iPhone 6 S", price: 54000), new Phone( name: "Lumia 950", price: 45000),
    new Phone( name: "Samsung Galaxy S 6", price: 40000));

phoneStream
    .flatMap(p->Stream.of(
        String.format("Назва: %s ціна без знижки: %d", p.getName(), p.getPrice()),
        String.format("Назва: %s ціна без знижки: %d", p.getName(), p.getPrice() - (int)(p.getPrice()*0.1))
    ))
    .forEach(System.out::println);
```

Сортування

Колекції, на основі яких нерідко створюються потоки, вже мають спеціальні методи для сортування вмісту. Однак клас **Stream** також включає можливість сортування. Таке сортування ми можемо задіяти, коли в нас іде набір проміжних операцій із потоком, які створюють нові набори даних, і нам треба ці набори відсортувати.

Для простого сортування за зростанням застосовується метод `sorted()`:

```
List<String> phones = new ArrayList<>();
Collections.addAll(phones, ...elements: "iPhone X", "Nokia 9", "Huawei Nexus 6P",
    "Samsung Galaxy S8", "LG G6", "Xiaomi MI6",
    "ASUS Zenfone 3", "Sony Xperia Z5", "Meizu Pro 6",
    "Pixel 2");

phones.stream()
    .filter(p->p.length()<12)
    .sorted() // сортування за зростанням
    .forEach(System.out::println);
```

Stream API

Однак цей метод не завжди підходить, метод сортує об'єкти за зростанням, але при цьому великі та малі літери розглядаються окремо. Крім того, цей метод підходить тільки для сортування тих об'єктів, які реалізують інтерфейс **Comparable**. Якщо ж у нас класи об'єктів не реалізують цей інтерфейс або ми хочемо створити якусь свою логіку сортування, то ми можемо використовувати іншу версію методу **sorted()**, яка як параметр приймає компаратор.

Наприклад, нехай у нас є такий клас **Phone**, відсортуємо потік об'єктів **Phone**:

```
class Phone{
    2 usages
    private String name;
    2 usages
    private String company;
    2 usages
    private int price;
    no usages
    public Phone(String name, String comp, int price){
        this.name=name;
        this.company=comp;
        this.price = price;
    }
    no usages
    public String getName() { return name; }
    no usages
    public int getPrice() { return price; }
    no usages
    public String getCompany() { return company; }
}
```

```
class PhoneComparator implements Comparator<Phone>{
    public int compare(Phone a, Phone b){
        return a.getName().toUpperCase().compareTo(b.getName().toUpperCase());
    }
}

class Program{
    public static void main(String[] args) {
        Stream<Phone> phoneStream = Stream.of(new Phone( name: "iPhone X", comp: "Apple", price: 600),
            new Phone( name: "Pixel 2", comp: "Google", price: 500),
            new Phone( name: "iPhone 8", comp: "Apple", price: 450),
            new Phone( name: "Nokia 9", comp: "HMD Global", price: 150),
            new Phone( name: "Galaxy S9", comp: "Samsung", price: 300));

        phoneStream.sorted(new PhoneComparator())
            .forEach(p->System.out.printf("%s (%s) - %d \n",
                p.getName(), p.getCompany(), p.getPrice()));
    }
}
```

Тут визначено клас компаратора **PhoneComparator**, який сортує об'єкти за полем name.

Stream API

Отримання підпотoku та об'єднання потоків

Низка методів Stream API повертають підпотoki або об'єднані потоки на основі вже наявних потоків. Розглянемо ці методи.

takeWhile

Метод **takeWhile()** вибирає з потоку елементи, поки вони відповідають умові. Якщо трапляється елемент, який не відповідає умові, то метод завершує свою роботу. Обрані елементи повертаються у вигляді потоку. В данном случае программа выбирает из потока числа, пока они меньше нуля.

```
Stream<Integer> numbers = Stream.of( ...values: -3, -2, -1, 0, 1, 2, 3, -4, -5);  
numbers.takeWhile(n -> n < 0)  
    .forEach(System.out::println);
```

При цьому незважаючи на те, що в потоці більше від'ємних чисел, але метод завершує роботу, щойно виявить перше число, яке не відповідає умові. У цьому і полягає відмінність, наприклад, від методу **filter()**.

Stream API

Щоб у цьому випадку охопити всі елементи, які менші за нуль, потік слід попередньо відсортувати:

```
Stream<Integer> numbers = Stream.of( ...values: -3, -2, -1, 0, 1, 2, 3, -4, -5);  
numbers.sorted().takeWhile(n -> n < 0)  
    .forEach(System.out::println);
```

dropWhile

Метод `dropWhile()` виконує зворотнє завдання - він пропускає елементи потоку, які відповідають умові доти, доки не зустрине елемент, який НЕ відповідає умові:

```
Stream<Integer> numbers = Stream.of( ...values: -3, -2, -1, 0, 1, 2, 3, -4, -5);  
numbers.sorted().dropWhile(n -> n < 0)  
    .forEach(System.out::println);
```

Stream API

concat

Статичний метод **concat()** об'єднує елементи двох потоків, повертаючи об'єднаний потік:

```
Stream<String> people1 = Stream.of( ...values: "Tom", "Bob", "Sam");  
Stream<String> people2 = Stream.of( ...values: "Alice", "Kate", "Sam");  
  
Stream.concat(people1, people2).forEach(System.out::println);
```

distinct

Метод **distinct()** повертає тільки унікальні елементи у вигляді потоку:

```
Stream<String> people = Stream.of( ...values: "Tom", "Bob", "Sam", "Tom", "Alice", "Kate", "Sam");  
people.distinct().forEach(System.out::println);
```

Stream API

Методи skip і limit

Метод **skip(long n)** використовується для пропуску n елементів. Цей метод повертає новий потік, у якому пропущено перші n елементів. Метод **limit(long n)** застосовується для вибірки перших n елементів потоків. Цей метод також повертає модифікований потік, у якому не більше n елементів. Найчастіше ця пара методів використовується разом для створення ефекту посторінкової навігації. Розглянемо, як їх застосовувати:

```
Stream<String> phoneStream = Stream.of(...values: "iPhone 6 S", "Lumia 950", "Samsung Galaxy S 6", "LG G 4", "Nexus 7");

phoneStream.skip(n: 1)
    .limit(maxSize: 2)
    .forEach(System.out::println);
```

У цьому випадку метод **skip** пропускає один перший елемент, а метод **limit** вибирає два наступні елементи.

Цілком може бути, що метод **skip** може приймати як параметр число більше, ніж кількість елементів у потоці. У цьому випадку будуть пропущені всі елементи, а в результуючому потоці буде 0 елементів. І якщо в метод **limit** передається число, більше, ніж кількість елементів, то просто вибираються всі елементи потоку. Тепер розглянемо, як створити посторінкову навігацію:

Stream API

```
List<String> phones = new ArrayList<>(Arrays.asList("iPhone 6 S", "Lumia 950", "Huawei Nexus 6P",  
    "Samsung Galaxy S 6", "LG G 4", "Xiaomi MI 5",  
    "ASUS Zenfone 2", "Sony Xperia Z5", "Meizu Pro 5",  
    "Lenovo S 850"));

int pageSize = 3; // кількість елементів на сторінку
Scanner scanner = new Scanner(System.in);
while(true){

    System.out.println("Введіть номер сторінки: ");
    int page = scanner.nextInt();

    if(page<1) break; // якщо число менше 1, виходимо з циклу

    phones.stream().skip((long) (page - 1) * pageSize)
        .limit(pageSize)
        .forEach(System.out::println);
}
```

У цьому випадку в нас набір із 10 елементів. За допомогою змінної `pageSize` визначаємо кількість елементів на сторінці - 3. Тобто в нас вийде **4 сторінки** (на останній буде тільки один елемент). У нескінченному циклі отримуємо номер сторінки та обираємо тільки ті елементи, які знаходяться на зазначеній сторінці.

Операції зведення

Операції зведення представляють термінальні операції, які повертають деяке значення - результат операції. У **Stream API** є низка операцій зведення. Метод **`count()`** повертає кількість елементів у потоці даних:

Stream API

```
ArrayList<String> names = new ArrayList<>(Arrays.asList(new String[]{"Tom", "Sam", "Bob", "Alice"}));
System.out.println(names.stream().count()); // 4

// кількість елементів із довжиною не більше 3 символів
System.out.println(names.stream().filter(n -> n.length() <= 3).count());
```

findFirst і findAny

Метод **findFirst()** витягує з потоку перший елемент, а **findAny()** витягує випадковий об'єкт із потоку (нерідко так само перший):

```
ArrayList<String> names = new ArrayList<>();
names.addAll(Arrays.asList("Tom", "Sam", "Bob", "Alice"));

Optional<String> first = names.stream().findFirst();
System.out.println(first.get()); // Tom

Optional<String> any = names.stream().findAny();
System.out.println(first.get()); // Tom
```

allMatch, anyMatch, noneMatch

Ще одна група операцій зведення повертає логічне значення **true** або **false**:

- **boolean allMatch(Predicate<? super T> predicate)**: повертає true, якщо всі елементи потоку задовольняють умові в предикаті;

Stream API

- **boolean anyMatch(Predicate<? super T> predicate):** повертає true, якщо хоч один елемент потоку задовольняє умову в предикаті;
- **boolean noneMatch(Predicate<? super T> predicate):** повертає true, якщо жоден з елементів у потоці не задовольняє умові в предикаті.

Приклад використання функцій:

```
ArrayList<String> names = new ArrayList<>();
names.addAll(Arrays.asList("Tom", "Sam", "Bob", "Alice"));

// чи є в потоці рядок, довжина якого більша за 3
boolean any = names.stream().anyMatch(s->s.length()>3);
System.out.println(any);    // true

// чи всі рядки мають довжину в 3 символи
boolean all = names.stream().allMatch(s->s.length()==3);
System.out.println(all);    // false

// чи НЕМАЄ в потоці рядка "Bill". Якщо немає, то true, якщо є, то false
boolean none = names.stream().noneMatch(s-> Objects.equals(s, "Bill"));
System.out.println(none);    // true
```

min і max

Методи **min()** і **max()** повертають відповідно мінімальне і максимальне значення. Оскільки дані в потоці можуть представляти різні типи, зокрема складні класи, то як параметр у ці методи передається об'єкт інтерфейсу **Comparator**, який вказує, як порівнювати об'єкти:

Stream API

`Optional<T> min(Comparator<? super T> comparator)`

`Optional<T> max(Comparator<? super T> comparator)`

Обидва методи повертають елемент потоку (мінімальний або максимальний), обгорнутий в об'єкт `Optional`. Наприклад, знайдемо мінімальне і максимальне число в числовому потоці:

```
ArrayList<Integer> numbers = new ArrayList<>();
numbers.addAll(Arrays.asList(1,2,3,4,5,6,7,8,9));

Optional<Integer> min = numbers.stream().min(Integer::compare);
Optional<Integer> max = numbers.stream().max(Integer::compare);
System.out.println(min.get()); // 1
System.out.println(max.get()); // 9
```

Інтерфейс `Comparator` - це функціональний інтерфейс, який визначає один метод `compare`, що приймає два об'єкти, які порівнюються, і повертає число (якщо перший об'єкт більший, повертається позитивне число, інакше повертається негативне число).

Тому замість конкретної реалізації компаратора ми можемо передати лямбда-вираз або метод, який відповідає методу `compare` інтерфейсу **`Comparator`**. Оскільки порівнюються числа, то в метод передається як компаратор статичний метод **`Integer.compare()`**. При цьому методи `min` і `max` повертають саме `Optional`, і щоб отримати безпосередньо результат операції з `Optional`, необхідно викликати метод `get()`.

Stream API

Метод reduce

Метод reduce виконує термінальні операції зведення, повертаючи деяке значення - результат операції. Він має такі форми:

`Optional<T> reduce(BinaryOperator<T> accumulator)`

`T reduce(T identity, BinaryOperator<T> accumulator)`

`U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner)`

```
Stream<Integer> numbersStream = Stream.of( ...values: 1,2,3,4,5,6);
Optional<Integer> result = numbersStream.reduce((x,y)->x*y);
System.out.println(result.get()); // 720
```

Об'єкт **BinaryOperator<T>** представляє функцію, яка приймає два елементи і виконує над ними деяку операцію, повертаючи результат. При цьому метод reduce зберігає результат і потім знову ж таки застосовує до цього результату і наступного елемента в наборі бінарну операцію. Фактично в цьому випадку ми отримаємо результат, який дорівнюватиме: $n_1 \text{ op } n_2 \text{ op } n_3 \text{ op } n_4 \text{ op } n_5 \text{ op } n_6$, де op - це операція (у даному випадку множення), а $n_1, n_2, \dots$ - елементи з потоку. Потім за допомогою методу `get()` ми можемо отримати власне результат обчислень: **result.get()**. Або ще один приклад - об'єднання слів у речення:

```
Stream<String> wordsStream = Stream.of( ...values: "Ревуть", "воли", "як", "ясла", "повні");
Optional<String> sentence = wordsStream.reduce((x,y)->x + " " + y);
System.out.println(sentence.get());
```

Тип Optional

Низка операцій відомості, такі як `min`, `max`, `reduce`, повертають об'єкт **Optional<T>**. Цей об'єкт фактично обгортає результат операції. Після виконання операції за допомогою методу **get()** об'єкта **Optional** ми можемо отримати його значення:

```
ArrayList<Integer> numbers = new ArrayList<>();  
numbers.addAll(Arrays.asList(1,2,3,4,5,6,7,8,9));  
Optional<Integer> min = numbers.stream().min(Integer::compare);  
System.out.println(min.get()); // 1
```

Але що, якщо потік не містить взагалі ніяких даних:

```
// список numbers порожній  
ArrayList<Integer> numbers = new ArrayList<>();  
Optional<Integer> min = numbers.stream().min(Integer::compare);  
System.out.println(min.get()); // java.util.NoSuchElementException
```

Тип Optional

У цьому випадку програма видасть виняток **java.util.NoSuchElementException**. Що ми можемо зробити, щоб уникнути викидання винятку? Для цього клас **Optional** надає низку методів. Найпростіший спосіб уникнути подібної ситуації - це попередня перевірка наявності значення в **Optional** за допомогою методу **isPresent()**. Він повертає **true**, якщо значення присутнє в **Optional**, і **false**, якщо значення відсутнє:

```
ArrayList<Integer> numbers = new ArrayList<>();  
Optional<Integer> min = numbers.stream().min(Integer::compare);  
if(min.isPresent()) System.out.println(min.get());
```

orElse

Метод **orElse()** дає змогу визначити альтернативне значення, яке буде повертатися, якщо **Optional** не отримає з потоку якого-небудь значення:

```
ArrayList<Integer> numbers = new ArrayList<>();  
Optional<Integer> min = numbers.stream().min(Integer::compare);  
System.out.println(min.orElse( other: -1)); // -1  
  
numbers.addAll(Arrays.asList(4,5,6,7,8,9));  
min = numbers.stream().min(Integer::compare);  
System.out.println(min.orElse( other: -1)); // 4
```

Тип Optional

orElseGet

Метод `orElseGet()` дає змогу задати функцію, яка повертатиме значення за замовчуванням:

```
ArrayList<Integer> numbers = new ArrayList<>();  
Optional<Integer> min = numbers.stream().min(Integer::compare);  
Random rnd = new Random();  
System.out.println(min.orElseGet(() -> rnd.nextInt( bound: 100)));
```

У цьому випадку значення, що повертається, генерується за допомогою методу **`nextInt`** класу **`Random`**, який повертає випадкове число.

orElseThrow

Ще один метод - **`orElseThrow`** дозволяє згенерувати виняток, якщо **`Optional`** не містить значення:

```
ArrayList<Integer> numbers = new ArrayList<>();  
Optional<Integer> min = numbers.stream().min(Integer::compare);  
// генерація виключення IllegalStateException  
System.out.println(min.orElseThrow(IllegalStateException::new));
```

Тип Optional

Обробка отриманого значення

Метод `ifPresent()` визначає дії зі значенням в `Optional`, якщо значення є:

```
ArrayList<Integer> numbers = new ArrayList<>();
numbers.addAll(Arrays.asList(4,5,6,7,8,9));
Optional<Integer> min = numbers.stream().min(Integer::compare);
min.ifPresent(System.out::println); // 4
```

У метод **`ifPresent`** передається функція, яка приймає один параметр - значення з **`Optional`**. У цьому випадку отримане мінімальне число виводиться на консоль. Але якби масив `numbers` був би порожнім, і відповідно **`Optional`** не стримав би жодного значення, то жодної помилки б не було.

Метод **`ifPresentOrElse()`** дає змогу визначити альтернативну логіку на випадок, якщо значення в `Optional` відсутнє:

```
ArrayList<Integer> numbers = new ArrayList<>();
Optional<Integer> min = numbers.stream().min(Integer::compare);
min.ifPresentOrElse(
    System.out::println,
    () -> System.out.println("Value not found")
);
```

Тип Optional

У метод **ifPresentOrElse** передається дві функції. Перша обробляє значення в Optional, якщо воно присутнє. Друга функція представляє дії, які виконуються, якщо значення в Optional відсутнє.

Метод collect

Більшість операцій класу Stream, які модифікують набір даних, повертають цей набір у вигляді потоку. Однак бувають ситуації, коли хотілося б отримати дані не у вигляді потоку, а у вигляді звичайної колекції, наприклад, **ArrayList** або **HashSet**.

І для цього у класу Stream визначено метод collect. Перша версія методу приймає як параметр функцію перетворення до колекції:

```
<R,A> R collect(Collector<? super T,A,R> collector)
```

Параметр R представляє тип результату методу, параметр T - тип елемента в потоці, а параметр A - тип проміжних накопичуваних даних. У підсумку параметр collector представляє функцію перетворення потоку в колекцію. Ця функція представляє об'єкт Collector, який визначено в пакеті **java.util.stream**. Ми можемо написати свою реалізацію функції, однак Java вже надає низку вбудованих функцій, визначених у класі Collectors:

- **toList()**: перетворення до типу **List**;
- **toSet()**: перетворення до типу **Set**;
- **toMap()**: перетворення до типу **Map**.

Наприклад, перетворюємо набір у потоці на список:

Тип Optional

```
List<String> phones = new ArrayList<>();
Collections.addAll(phones, "iPhone 8", "HTC U12", "Huawei Nexus 6P",
    "Samsung Galaxy S9", "LG G6", "Xiaomi MI6", "ASUS Zenfone 2",
    "Sony Xperia Z5", "Meizu Pro 6", "Lenovo S850");

List<String> filteredPhones = phones.stream()
    .filter(s->s.length()<10)
    .toList();

for(String s : filteredPhones)
    System.out.println(s);
```

Використання методу **toSet()** аналогічне.

```
Set<String> filteredPhones = phones.stream()
    .filter(s->s.length()<10)
    .collect(Collectors.toSet());
```

Групування

Щоб згрупувати дані за якою-небудь ознакою, нам треба використовувати у зв'язці метод **collect()** об'єкта **Stream** і метод **Collectors.groupingBy()**. Припустимо, у нас є такий клас:

Тип Optional

І, наприклад, у нас є набір об'єктів **Phone**, які ми хочемо згрупувати за компанією:

```
class Phone{  
    2 usages  
    private String name;  
    2 usages  
    private String company;  
    2 usages  
    private int price;  
  
    no usages  
    public Phone(String name, String comp, int price){  
        this.name=name;  
        this.company=comp;  
        this.price = price;  
    }  
  
    no usages  
    public String getName() { return name; }  
    no usages  
    public int getPrice() { return price; }  
    no usages  
    public String getCompany() { return company; }  
}
```

```
Stream<Phone> phoneStream = Stream.of(new Phone( name: "iPhone X", comp: "Apple", price: 600),  
    new Phone( name: "Pixel 2", comp: "Google", price: 500),  
    new Phone( name: "iPhone 8", comp: "Apple", price: 450),  
    new Phone( name: "Galaxy S9", comp: "Samsung", price: 440),  
    new Phone( name: "Galaxy S8", comp: "Samsung", price: 340));  
  
Map<String, List<Phone>> phonesByCompany = phoneStream.collect(  
    Collectors.groupingBy(Phone::getCompany));  
  
for(Map.Entry<String, List<Phone>> item : phonesByCompany.entrySet()){  
    System.out.println(item.getKey());  
    for(Phone phone : item.getValue())  
        System.out.println(phone.getName());  
  
    System.out.println();  
}
```

Отже, для створення груп у метод **phoneStream.collect()** передається виклик функції **Collectors.groupingBy()**, яка за допомогою виразу **Phone::getCompany** групує об'єкти за компанією. У підсумку буде створено об'єкт Map, у якому ключами є назви компаній, а значеннями - список пов'язаних із компаніями телефонів.

Тип Optional

Метод **Collectors.partitioningBy**

Метод **Collectors.partitioningBy()** має схожу дію, тільки він ділить елементи на групи за принципом, чи відповідає елемент певній умові. Наприклад:

```
Stream<Phone> phoneStream = Stream.of(new Phone( name: "iPhone X", comp: "Apple", price: 600),
    new Phone( name: "Pixel 2", comp: "Google", price: 500),
    new Phone( name: "iPhone 8", comp: "Apple", price: 450),
    new Phone( name: "Galaxy S9", comp: "Samsung", price: 440),
    new Phone( name: "Galaxy S8", comp: "Samsung", price: 340));

Map<Boolean, List<Phone>> phonesByCompany = phoneStream.collect(
    Collectors.partitioningBy(p-> Objects.equals(p.getCompany(), b: "Apple")));

for(Map.Entry<Boolean, List<Phone>> item : phonesByCompany.entrySet()){
    System.out.println(item.getKey());
    for(Phone phone : item.getValue()){
        System.out.println(phone.getName());
    }

    System.out.println();
}
```

У цьому випадку за допомогою умови **p->p.getCompany()=="Apple"** ми дивимося, чи належить телефон компанії Apple. Якщо телефон належить цій компанії, то він потрапляє в одну групу, якщо ні, то в іншу.

Тип Optional

Метод **Collectors.counting**

Метод **Collectors.counting** застосовується в **Collectors.groupingBy()** для обчислення кількості елементів у кожній групі:

```
Map<String, Long> phonesByCompany = phoneStream.collect(
    Collectors.groupingBy(Phone::getCompany, Collectors.counting()));

for(Map.Entry<String, Long> item : phonesByCompany.entrySet())
    System.out.println(item.getKey() + " - " + item.getValue());
```

Метод **Collectors.summing**

Метод **Collectors.summing** застосовується для підрахунку суми. Залежно від типу даних, до яких застосовується метод, він має такі форми: **summingInt()**, **summingLong()**, **summingDouble()**. Застосуємо цей метод для підрахунку вартості всіх смартфонів за компаніями:

```
Map<String, Integer> phonesByCompany = phoneStream.collect(
    Collectors.groupingBy(Phone::getCompany, Collectors.summingInt(Phone::getPrice)));

for(Map.Entry<String, Integer> item : phonesByCompany.entrySet())
    System.out.println(item.getKey() + " - " + item.getValue());
```

Тип Optional

За допомогою виразу **Collectors.summingInt(Phone::getPrice))** ми вказуємо, що для кожної компанії обчислюватиметься сукупна ціна всіх її смартфонів. І оскільки обчислюється результат - сума для значень типу `int`, то як тип колекції, що повертається, використовується тип `Map<String, Integer>`

Методи `maxBy` та `minBy`

Методи **`maxBy`** і **`minBy`** застосовуються для підрахунку мінімального і максимального значення в кожній групі. Як параметр ці методи приймають функцію компаратора, яка потрібна для порівняння значень. Наприклад, знайдемо для кожної компанії телефон з мінімальною ціною:

```
Map<String, Optional<Phone>> phonesByCompany = phoneStream.collect(
    Collectors.groupingBy(Phone::getCompany,
        Collectors.minBy(Comparator.comparing(Phone::getPrice))));

for(Map.Entry<String, Optional<Phone>> item : phonesByCompany.entrySet())
    System.out.println(item.getKey() + " - " + item.getValue().get().getName());
```

Тип Optional

Як значення операції групування, що повертається, використовується об'єкт **Map<String, Optional<Phone>>**. Знову ж таки оскільки групуємо за компаніями, то ключем буде виступати рядок, а значенням - об'єкт **Optional<Phone>**.

Метод summarizing

Методи **summarizingInt()** / **summarizingLong()** / **summarizingDouble()** дають змогу об'єднати в набір значення відповідних типів:

```
Map<String, IntSummaryStatistics> priceSummary = phoneStream.collect(
    Collectors.groupingBy(Phone::getCompany,
        Collectors.summarizingInt(Phone::getPrice)));

for(Map.Entry<String, IntSummaryStatistics> item : priceSummary.entrySet())
    System.out.println(item.getKey() + " - " + item.getValue().getAverage());
```

Тип Optional

Метод **Collectors.summarizingInt(Phone::getPrice)** створює набір, у який поміщаються ціни для всіх телефонів кожної з груп. Цей набір інкапсулюється в об'єкті **IntSummaryStatistics**. Відповідно якби ми застосовували методи **summarizingLong()** або **summarizingDouble()**, то відповідно б отримували об'єкти **LongSummaryStatistics** або **DoubleSummaryStatistics**.

У цих об'єктів є низка методів, які дозволяють виконати різні атомарні операції над набором:

- **getAverage()**: повертає середнє значення;
- **getCount()**: повертає кількість елементів у наборі;
- **getMax()**: повертає максимальне значення;
- **getMin()**: повертає мінімальне значення;
- **getSum()**: повертає суму елементів;
- **accept()**: додає в набір новий елемент

У цьому випадку ми отримуємо середню ціну смартфонів для кожної групи.

Паралельні потоки

Паралельні потоки

Крім послідовних потоків **Stream API** підтримує паралельні потоки. Розпаралелювання потоків дає змогу задіяти кілька ядер процесора (якщо цільова машина багатоядерна) і тим самим може підвищити продуктивність і прискорити обчислення. Водночас говорити, що застосування паралельних потоків на багатоядерних машинах однозначно підвищить продуктивність - не зовсім коректно. У кожному конкретному випадку треба перевіряти і тестувати.

Щоб зробити звичайний послідовний потік паралельним, треба викликати в об'єкта **Stream** метод **parallel**. Крім того, можна також використовувати метод **parallelStream()** інтерфейсу **Collection** для створення паралельного потоку з колекції.

Водночас якщо робоча машина не є багатоядерною, то потік виконуватиметься як послідовний.

Застосування паралельних потоків у багатьох випадках буде аналогічним. Нижче приведено два приклади

```
Stream<Integer> numbersStream = Stream.of(1, 2, 3, 4, 5, 6);
Optional<Integer> result = numbersStream.parallel().reduce((x,y)-> x*y);
System.out.println(result.get()); // 720
```

```
List<String> people = Arrays.asList("Tom", "Bob", "Sam", "Kate", "Tim");

System.out.println("Послідовний потік");
people.stream().filter(p->p.length()==3).forEach(System.out::println);

System.out.println("\nПаралельний потік");
people.parallelStream().filter(p->p.length()==3).forEach(System.out::println);
```

Паралельні потоки

У цьому випадку спочатку для списку **people** створюємо потік і виконуємо над ним низку операцій у послідовному режимі. Зокрема, знаходимо в списку рядки, довжина яких дорівнює 3 і виводимо їх на консоль. У цьому разі всі операції з потоком будуть виконуватися над елементами списку в тому порядку, в якому елементи йдуть у списку. Потім за допомогою методу **people.parallelStream()** для списку створюється паралельний потік. Причому застосовуються ті самі операції, однак тепер порядок, у якому над елементами списку будуть проводитися операції, не детермінований.

Паралельні потоки

У випадку з паралельним потоком висновок недетермінований і може відрізнятись. Однак не всі функції можна без шкоди для точності обчислення перенести з послідовних потоків на паралельні. Насамперед такі функції мають бути без збереження стану та асоціативними, тобто при виконанні зліва направо давати той самий результат, що й при виконанні справа наліво, як у випадку з добутком чисел. Наприклад:

```
Stream<Integer> numbersStream = Stream.of(1, 2, 3, 4, 5, 6);  
Integer result = numbersStream.parallel().reduce(1, (x,y)->x * y);  
System.out.println(result);
```

Фактично тут відбувається перемноження чисел. При цьому немає різниці між $1*2*3*4*(5*6)$ або $5*6*1*(2*3)*4$. Ми можемо розставити дужки будь-яким чином, розмістити послідовність чисел у будь-якому порядку, і все одно ми отримаємо один і той самий результат. Тобто ця операція є асоціативною і тому може бути розпаралелена.

Паралельні потоки

Питання продуктивності в паралельних операціях

Фактично застосування паралельних потоків зводиться до того, що дані в потоці будуть розділені на частини, кожна частина обробляється на окремому ядрі процесора, і врешті ці частини з'єднуються, і над ними виконуються фінальні операції. Розглянемо пару критеріїв, які можуть вплинути на продуктивність у паралельних потоках:

- **Розмір даних.** Що більші дані, то складніше спочатку розділяти дані, а потім їх з'єднувати.
- **Кількість ядер процесора.** Теоретично, чим більше ядер у комп'ютері, тим швидше програма працюватиме. Якщо на машині одне ядро, немає сенсу застосовувати паралельні потоки.

Що простіша структура даних, з якою працює потік, то швидше відбуватимуться операції. Наприклад, дані з **ArrayList** легко використовувати, оскільки структура цієї колекції передбачає послідовність незв'язаних даних. А ось колекція типу **LinkedList** - не найкращий варіант, оскільки в послідовному списку всі елементи пов'язані з попередніми/наступними. І такі дані важко розпаралелити. Над даними примітивних типів операції будуть проводитися швидше, ніж над об'єктами класів.

Контрольні питання

1. Охарактеризуйте поняття лямбда-виразу в контексті програмування на Java.
2. Задайте визначення поняттю функціонального інтерфейсу та поясніть його значення в мові програмування Java.
3. Що представляє собою Stream API в мові програмування Java? Як він покращує роботу з даними в Java?
4. Які є найпопулярніші методи при роботі з Stream API у мові програмування Java?
5. Для чого використовується тип Optional в мові програмування Java?
6. Чи можете пояснити різницю між колекцією та потоком в контексті Java?
7. Чи можливо перетворити масив на потік в Java?
8. Перерахуйте вбудовані функціональні інтерфейси в мові програмування Java.

The END
Lec 11.

.NET та JAVA #12

Багатопотокове програмування

1. Введення в багатопотокове програмування. Клас Thread
2. Створення, виконання, завершення та переривання потоків
3. Синхронізація потоків. Оператор synchronized
4. Взаємодія потоків. Методи wait та notify
5. Семафори
6. Обмін між потоками. Клас Exchanger
7. Клас Phaser
8. Блокування. ReentrantLock
9. Умови блокування

Введення в багатопотокове програмування.

Клас Thread

Більшість мов програмування підтримують таку важливу функціональність як **багатопоточність**, і **Java** в цьому плані не виняток. За допомогою багатопоточності ми можемо виділити в додатку кілька потоків, які виконуватимуть різні завдання одночасно. Якщо у нас, припустімо, графічний застосунок, який надсилає запит до якогось сервера або зчитує й обробляє величезний файл, то без багатопотоковості у нас би блокувався графічний інтерфейс на час виконання завдання. А завдяки потокам ми можемо виділити надсилання запиту або будь-яке інше завдання, яке може довго оброблятися, в окремий потік. Тому більшість реальних додатків, які багатьом із нас доводиться використовувати, практично не мислимі без багатопоточності.

У Java функціональність окремого потоку полягає в класі **Thread**. І щоб створити новий потік, нам треба створити об'єкт цього класу. Але всі потоки не створюються самі по собі. Коли запускається програма, починає працювати головний потік цієї програми. Від цього головного потоку породжуються всі інші дочірні потоки. За допомогою статичного методу **Thread.currentThread()** ми можемо отримати поточний потік виконання:

```
class Program {  
    public static void main(String[] args) {  
        Thread t = Thread.currentThread(); // отримуємо головний потік  
        System.out.println(t.getName()); // main  
    }  
}
```

Введення в багатопотокове програмування.

Клас Thread

За замовчуванням іменем головного потоку буде **main**. Для керування потоком клас **Thread** надає ще низку методів. Найбільш використовувані з них:

- **getName():** повертає ім'я потоку;
- **setName(String name):** встановлює ім'я потоку;
- **getPriority():** повертає пріоритет потоку;
- **setPriority(int priority):** встановлює пріоритет потоку. Пріоритет є одним із ключових факторів для вибору системою потоку з купи потоків для виконання. У цей метод як параметр передається числове значення пріоритету - від 1 до 10. За замовчуванням головному потоку виставляється середній пріоритет – 5;
- **isAlive():** повертає true, якщо потік активний;
- **isInterrupted():** повертає true, якщо потік був перерваний;
- **join():** очікує завершення потоку;
- **run():** визначає точку входу в потік;
- **sleep():** призупиняє потік на задану кількість мілісекунд;
- **start():** запускає потік, викликаючи його метод run().

Введення в багатопотокове програмування.

Клас Thread

Ми можемо вивести всю інформацію про потік:

```
class Program {  
    public static void main(String[] args) {  
        Thread t = Thread.currentThread(); // отримуємо головний потік  
        System.out.println(t); // Thread[#1,main,5,main]  
    }  
}
```

Перше `main` представлятиме ім'я потоку (що можна отримати через **`t.getName()`**), друге значення `5` надає пріоритет потоку (також можна отримати через **`t.getPriority()`**), і останнє `main` представляє ім'я групи потоків, до якого належить поточний - за замовчуванням також `main` (також можна отримати через **`t.getThreadGroup().getName()`**).

Введення в багатопотокове програмування.

Клас Thread

Недоліки під час використання потоків

Далі ми розглянемо, як створювати та використовувати потоки. Це досить легко. Однак при створенні багатопотокового додатка нам слід враховувати низку обставин, які негативно можуть позначитися на роботі програми. На деяких платформах запуск нових потоків може уповільнити роботу програми. Що може мати велике значення, якщо нам критична продуктивність програми. Для кожного потоку створюється свій власний стек у пам'яті, куди поміщаються всі локальні змінні та низка інших даних, пов'язаних із виконанням потоку.

Відповідно, **чим більше потоків створюється, тим більше пам'яті використовується**. При цьому треба пам'ятати, у будь-якій системі розміри використовуваної пам'яті обмежені. Крім того, у багатьох системах може бути обмеження на кількість потоків. Але навіть якщо такого обмеження немає, то в будь-якому разі є природне обмеження у вигляді максимальної швидкості процесора.

Створення, виконання, завершення та переривання потоків

Для створення нового потоку ми можемо створити новий клас, або успадковуючи його від класу **Thread**, або реалізуючи в класі інтерфейс **Runnable**.

Успадкування від класу Thread

Створимо свій клас на основі **Thread**:

```
class MyThread extends Thread {
    1 usage
    MyThread(String name){
        super(name);
    }
    public void run(){
        System.out.printf("%s started... \n", Thread.currentThread().getName());
        try{
            Thread.sleep( 1 millis: 500);
        }
        catch(InterruptedException e){
            System.out.println("Thread has been interrupted");
        }
        System.out.printf("%s finished... \n", Thread.currentThread().getName());
    }
}

class Program {
    public static void main(String[] args) {
        System.out.println("Main thread started...");
        new MyThread( name: "MyThread").start();
        System.out.println("Main thread finished...");
    }
}
```

```
Main thread started...
Main thread finished...
MyThread started...
MyThread finished...

Process finished with exit code 0
```

Створення, виконання, завершення та переривання потоків

Клас потоку називається **MyThread**. Передбачається, що в конструктор класу передається ім'я потоку, яке потім передається в конструктор базового класу. У конструктор свого класу потоку ми можемо передати різні дані, але головне, щоб у ньому викликався конструктор базового класу **Thread**, у який передається ім'я потоку. І також у **MyThread** перевизначається метод `run()`, код якого, власне, і буде представляти весь той код, який виконується в потоці. У методі `main` для запуску потоку **MyThread** у нього викликається метод `start()`, після чого починається виконання того коду, який визначено в методі `run`:

```
new MyThread("MyThread").start();
```

Тут у методі `main` у конструктор **MyThread** передається довільна назва потоку, і потім викликається метод `start()`. По суті цей метод якраз і викликає перевизначений метод `run()` класу **MyThread**.

Створення, виконання, завершення та переривання потоків

Аналогічно створенню одного потоку ми можемо запускати відразу кілька потоків:

```
public static void main(String[] args) {  
    System.out.println("Main thread started...");  
    for(int i = 1; i < 6; i++)  
        new MyThread( name: "MyThread " + i).start();  
    System.out.println("Main thread finished...");  
}
```

```
Main thread started...  
MyThread 1 started...  
MyThread 2 started...  
Main thread finished...  
MyThread 3 started...  
MyThread 4 started...  
MyThread 5 started...  
MyThread 3 finished...  
MyThread 2 finished...  
MyThread 1 finished...  
MyThread 4 finished...  
MyThread 5 finished...  
  
Process finished with exit code 0
```

Очікування завершення потоку

Під час запуску потоків у прикладах вище Main thread завершувався до дочірнього потоку. Як правило, більш поширеною ситуацією є випадок, коли Main thread завершується найостаннішим. Для цього треба застосувати метод **join()**. У цьому випадку поточний потік очікуватиме завершення потоку, для якого викликано метод join:

Створення, виконання, завершення та переривання потоків

```
public static void main(String[] args) {  
    System.out.println("Main thread started...");  
    MyThread t= new MyThread( name: "MyThread ");  
    t.start();  
    try{  
        t.join();  
    }  
    catch(InterruptedException e){  
        System.out.printf("%s has been interrupted", t.getName());  
    }  
    System.out.println("Main thread finished...");  
}
```

```
Main thread started...  
MyThread started...  
MyThread finished...  
Main thread finished...  
  
Process finished with exit code 0
```

Метод `join()` змушує потік, який викликав потік (у цьому випадку Main thread), очікувати завершення потоку, що викликається, для якого і застосовується метод `join` (у цьому випадку **MyThread**).

Якщо в програмі використовується кілька дочірніх потоків, і треба, щоб Main thread завершувався після дочірніх, то для кожного дочірнього потоку треба викликати метод `join`.

Створення, виконання, завершення та переривання потоків

Реалізація інтерфейсу Runnable

Інший спосіб визначення потоку представляє реалізація інтерфейсу **Runnable**. Цей інтерфейс має один метод run:

```
interface Runnable{  
    void run();  
}
```

У методі run() власне визначається весь той код, який виконується під час запуску потоку. Після визначення об'єкта **Runnable** він передається в один із конструкторів класу **Thread**:

Thread(Runnable runnable, String threadName)

Для реалізації інтерфейсу визначимо такий клас **MyThread**:

```
class MyThread implements Runnable {  
    public void run(){  
        System.out.printf("%s started... \n", Thread.currentThread().getName());  
        try{  
            Thread.sleep( (mins: 500);  
        }  
        catch(InterruptedException e){  
            System.out.println("Thread has been interrupted");  
        }  
        System.out.printf("%s finished... \n", Thread.currentThread().getName());  
    }  
}  
  
class Program {  
    public static void main(String[] args) {  
        System.out.println("Main thread started...");  
        Thread myThread = new Thread(new MyThread(), name: "MyThread");  
        myThread.start();  
        System.out.println("Main thread finished...");  
    }  
}
```

```
Main thread started...  
Main thread finished...  
MyThread started...  
MyThread finished...
```

Створення, виконання, завершення та переривання потоків

Реалізація інтерфейсу **Runnable** багато в чому аналогічна перевизначенню класу **Thread**. Також у методі `run` визначається найпростіший код, який присипляє потік на 500 мілісекунд. У методі `main` викликається конструктор **Thread**, у який передається об'єкт **MyThread**. І щоб запустити потік, викликається метод `start()`.

Оскільки **Runnable** фактично представляє функціональний інтерфейс, який визначає один метод, то об'єкт цього інтерфейсу ми можемо представити у вигляді лямбда-виразу:

```
System.out.println("Main thread started...");
Runnable r = ()->{
    System.out.printf("%s started... \n", Thread.currentThread().getName());
    try{
        Thread.sleep( millis: 500);
    }
    catch(InterruptedException e){
        System.out.println("Thread has been interrupted");
    }
    System.out.printf("%s finished... \n", Thread.currentThread().getName());
};
Thread myThread = new Thread(r, name: "MyThread");
myThread.start();
System.out.println("Main thread finished...");
```

```
Main thread started...
Main thread finished...
MyThread started...
MyThread finished...
```

Створення, виконання, завершення та переривання потоків

Завершення та переривання потоку

Приклади потоків раніше представляли потік як послідовний набір операцій. Після виконання останньої операції завершувався і потік. Однак нерідко має місце й інша організація потоку у вигляді нескінченного циклу. Наприклад, потік сервера в нескінченному циклі прослуховує певний порт на предмет отримання даних. І в цьому випадку ми також можемо передбачити механізм завершення потоку.

Завершення потоку

Поширений спосіб завершення потоку являє собою опитування логічної змінної. І якщо вона дорівнює, наприклад, `false`, то потік завершує нескінченний цикл і закінчує своє виконання. Визначимо наступний клас потоку:

Створення, виконання, завершення та переривання потоків

```
class MyThread implements Runnable {
    3 usages
    private boolean isActive;
    no usages
    void disable(){isActive=false;}
    no usages
    MyThread(){isActive = true;}
    public void run(){
        System.out.printf("%s started... \n", Thread.currentThread().getName());
        int counter=1; // лічильник циклів
        while(isActive){
            System.out.println("Loop " + counter++);
            try{
                Thread.sleep(400);
            }
            catch(InterruptedException e){
                System.out.println("Thread has been interrupted");
            }
        }
        System.out.printf("%s finished... \n", Thread.currentThread().getName());
    }
}
```

```
System.out.println("Main thread started...");
MyThread myThread = new MyThread();
new Thread(myThread, "MyThread").start();
try{
    Thread.sleep(1100);
    myThread.disable();
    Thread.sleep(1000);
}
catch(InterruptedException e){
    System.out.println("Thread has been interrupted");
}
System.out.println("Main thread finished...");
```

```
Main thread started...
MyThread started...
Loop 1
Loop 2
Loop 3
MyThread finished...
Main thread finished...
```

```
Process finished with exit code 0
```

Змінна **isActive** вказує на активність потоку. За допомогою методу `disable()` ми можемо скинути стан цієї змінної. Використаємо клас. Отже, спочатку запускається дочірній потік: `new Thread(myThread, "MyThread").start()`. Потім на 1100 мілісекунд зупиняємо Main thread і потім викликаємо метод **`myThread.disable()`**, який перемикає в потоці прапор `isActive`. І дочірній потік завершується.

Створення, виконання, завершення та переривання потоків

Метод interrupt

Ще один спосіб виклику завершення або переривання потоку представляє метод **interrupt()**. Виклик цього методу встановлює у потоку статус, що він перерваний. Сам метод повертає true, якщо потік може бути перерваний, в іншому разі повертається false. При цьому сам виклик цього методу НЕ завершує потік, він тільки встановлює статус: зокрема, метод **isInterrupted()** класу **Thread** повертатиме значення true. Ми можемо перевірити значення, що повертається даним методом, і виконати деякі дії. Приклад приведено на наступному слайді.

```
class MyThread extends Thread {
    1 usage
    MyThread(String name){super(name);}
    public void run(){
        System.out.printf("%s started... \n", Thread.currentThread().getName());
        int counter=1; // лічильник циклів
        while(!isInterrupted())
            System.out.println("Loop " + counter++);
        System.out.printf("%s finished... \n", Thread.currentThread().getName());
    }
}

class Program {
    public static void main(String[] args) {
        System.out.println("Main thread started...");
        MyThread t = new MyThread("JThread");
        t.start();
        try{
            Thread.sleep(150);
            t.interrupt();
            Thread.sleep(150);
        }
        catch(InterruptedException e){
            System.out.println("Thread has been interrupted");
        }
        System.out.println("Main thread finished...");
    }
}
```

```
Main thread started...
JThread started...
Loop 1
Loop 2
Loop 3
Loop 4
Loop 5
Loop 6
Loop 7
Loop 8
Loop 9
Loop 10
Loop 11
Loop 12
```

Створення, виконання, завершення та переривання потоків

У класі, який успадкований від **Thread**, ми можемо отримати статус поточного потоку за допомогою методу **isInterrupted()**. І поки цей метод повертає false, ми можемо виконувати цикл. А після того, як буде викликано метод **interrupt**, **isInterrupted()** поверне true, і відповідно відбудеться вихід із циклу.

Якщо основна функціональність укладена в класі, який реалізує інтерфейс **Runnable**, то там можна перевіряти статус потоку за допомогою методу **Thread.currentThread().isInterrupted()**:

```
class MyThread implements Runnable {
    public void run(){
        System.out.printf("%s started... \n", Thread.currentThread().getName());
        int counter=1; // лічильник циклів
        while(!Thread.currentThread().isInterrupted())
            System.out.println("Loop " + counter++);
        System.out.printf("%s finished... \n", Thread.currentThread().getName());
    }
}

class Program {
    public static void main(String[] args) {
        System.out.println("Main thread started...");
        MyThread myThread = new MyThread();
        Thread t = new Thread(myThread, "MyThread");
        t.start();
        try{
            Thread.sleep(150);
            t.interrupt();
            Thread.sleep(150);
        }
        catch(InterruptedException e){
            System.out.println("Thread has been interrupted");
        }
        System.out.println("Main thread finished...");
    }
}
```

Однак під час отримання статусу потоку за допомогою методу **isInterrupted()** слід враховувати, що якщо ми обробляємо в циклі виняток **InterruptedException** у блоці catch, то під час перехоплення винятку статус потоку автоматично скидається, і після цього **isInterrupted** повертатиме false.

Створення, виконання, завершення та переривання потоків

Наприклад, додамо в цикл потоку затримку за допомогою методу `sleep`:

```
public void run(){
    System.out.printf("%s started... \n", Thread.currentThread().getName());
    int counter=1; // лічильник циклів
    while(!Thread.currentThread().isInterrupted()){
        System.out.println("Loop " + counter++);
        try{
            Thread.sleep(100);
        }
        catch(InterruptedException e){
            System.out.println(Thread.currentThread().getName() + " has been interrupted");
            System.out.println(Thread.currentThread().isInterrupted()); // false
            Thread.currentThread().interrupt(); // повторно скидаємо стан
        }
    }
    System.out.printf("%s finished... \n", Thread.currentThread().getName());
}
```

Коли потік викличе метод `interrupt`, метод `sleep` згенерує виняток **`InterruptedException`**, і управління перейде до блоку `catch`. Але якщо ми перевіримо статус потоку, то побачимо, що метод **`isInterrupted`** повертає `false`. Як варіант, у цьому випадку ми можемо повторно перервати поточний потік, знову ж таки викликавши метод **`interrupt()`**. Тоді під час нової ітерації циклу `while` методу `isInterrupted` поверне `true`, і відбудеться вихід із циклу.

Синхронізація потоків. Оператор synchronized

Під час роботи потоки нерідко звертаються до якихось **спільних ресурсів**, які визначені поза потоком, наприклад, звернення до якогось файлу.

Якщо одночасно кілька потоків звернуться до спільного ресурсу, то результати виконання програми можуть бути несподіваними і навіть непередбачуваними. Наприклад, визначимо такий код:

```
class CommonResource{
    3 usages
    int x=0;
}

1 usage
class CountThread implements Runnable{
    4 usages
    CommonResource res;
    1 usage
    CountThread(CommonResource res){this.res=res;}
    public void run(){
        res.x=1;
        for (int i = 1; i < 5; i++){
            System.out.printf("%s %d \n", Thread.currentThread().getName(), res.x);
            res.x++;
            try{
                Thread.sleep(100);
            }
            catch(InterruptedException e){}
        }
    }
}
```

```
public static void main(String[] args) {
    CommonResource commonResource= new CommonResource();
    for (int i = 1; i < 6; i++){
        Thread t = new Thread(new CountThread(commonResource));
        t.setName("Thread "+ i);
        t.start();
    }
}
```

```
Thread 1 1
Thread 2 1
Thread 3 1
Thread 4 1
Thread 5 1
Thread 5 6
Thread 3 6
Thread 1 6
Thread 2 6
Thread 4 6
Thread 3 11
Thread 5 11
```

Синхронізація потоків. Оператор `synchronized`

Тут визначено клас **`CommonResource`**, який представляє загальний ресурс і в якому визначено одне цілочисельне поле `x`. Цей ресурс використовується класом потоку **`CountThread`**. Цей клас просто збільшує в циклі значення `x` на одиницю. Причому при вході в потік значення `x=1`:

```
res.x=1
```

Тобто в підсумку ми очікуємо, що після виконання циклу `res.x` дорівнюватиме 4. У головному класі програми запускається п'ять потоків. Тобто ми очікуємо, що кожен потік буде збільшувати `res.x` з 1 до 4 і так п'ять разів. Але якщо ми подивимося на результат роботи програми, приведений на минулому слайді, то він буде іншим. Тобто поки один потік не закінчив роботу з полем `res.x`, з ним починає працювати інший потік.

Синхронізація потоків. Оператор `synchronized`

Щоб уникнути подібної ситуації, треба синхронізувати потоки. Одним зі способів синхронізації є використання ключового слова ***synchronized***. Цей оператор передує блоку коду або методу, який підлягає синхронізації. Для його застосування змінимо клас **CountThread**:

```
class CountThread implements Runnable{
    5 usages
    CommonResource res;
    1 usage
    CountThread(CommonResource res){this.res=res;}
    public void run(){
        synchronized(res){
            res.x=1;
            for (int i = 1; i < 5; i++){
                System.out.printf("%s %d \n", Thread.currentThread().getName(), res.x);
                res.x++;
                try{
                    Thread.sleep( millis: 100);
                }
                catch (InterruptedException e){}
            }
        }
    }
}
```

При створенні синхронізованого блоку коду після оператора ***synchronized*** йде об'єкт-заглушка: `synchronized(res)`. Причому як об'єкт може використовуватися тільки об'єкт якого-небудь класу, але не примітивного типу.

Синхронізація потоків. Оператор `synchronized`

Кожен об'єкт у Java має асоційований із ним **монітор**. Монітор являє собою своєрідний інструмент для управління доступом до об'єкта. Коли виконання коду доходить до оператора ***synchronized***, монітор об'єкта `res` блокується, і на час його блокування монопольний доступ до блоку коду має тільки один потік, який і зробив блокування. Після закінчення роботи блоку коду, монітор об'єкта **`res`** звільняється і стає доступним для інших потоків.

Після звільнення монітора його захоплює інший потік, а всі інші потоки продовжують очікувати його звільнення.

У підсумку консольний вивід зміниться:

```
Thread 1 1  
Thread 1 2  
Thread 1 3  
Thread 1 4  
Thread 5 1  
Thread 5 2  
Thread 5 3  
Thread 5 4
```

Синхронізація потоків. Оператор *synchronized*

При застосуванні оператора ***synchronized*** до методу, поки цей метод не завершить виконання, монопольний доступ має тільки один потік - перший, який почав його виконання. Для застосування *synchronized* до методу, змінимо клас **CommonResource**:

```
class CommonResource{
    6 usages
    int x;
    no usages
    synchronized void increment(){
        x=1;
        for (int i = 1; i < 5; i++){
            System.out.printf("%s %d \n", Thread.currentThread().getName(), x);
            x++;
            try{
                Thread.sleep(100);
            }
            catch(InterruptedException e){}
        }
    }
}
```

Результат роботи в цьому разі буде аналогічний прикладу вище з блоком ***synchronized***. Тут знову в справу вступає монітор об'єкта **CommonResource** - спільного об'єкта для всіх потоків.

Взаємодія потоків. Методи `wait` та `notify`

Іноді під час взаємодії потоків постає питання про сповіщення одних потоків про дії інших. Наприклад, дії одного потоку залежать від результату дій іншого потоку, і треба якось сповістити один потік, що другий потік виконав якусь роботу.

І для подібних ситуацій у класу **Object** визначено низку методів:

- **`wait()`**: звільняє монітор і переводить потік, що викликає, у стан очікування доти, доки інший потік не викличе метод **`notify()`**;
- **`notify()`**: продовжує роботу потоку, у якого раніше було викликано метод `wait()`;
- **`notifyAll()`**: відновлює роботу всіх потоків, у яких раніше був викликаний метод `wait()`.

Усі ці методи викликаються тільки з синхронізованого контексту - синхронізованого блоку або методу.

Взаємодія потоків. Методи wait та notify

Розглянемо, як ми можемо використовувати ці методи. Візьмемо стандартну задачу з минулої теми - **"Виробник-Споживач" ("Producer-Consumer")**: поки виробник не виробив продукт, споживач не може його купити. Нехай виробник має виробити 5 товарів, відповідно споживач має їх усі купити. Але при цьому одночасно на складі може перебувати не більше 3 товарів. Для вирішення цього завдання задіємо методи **wait()** і **notify()**:

```
class Producer implements Runnable{  
    2 usages  
    Store store;  
    1 usage  
    Producer(Store store){this.store=store;}  
    public void run(){  
        for (int i = 1; i < 6; i++){  
            store.put();  
        }  
    }  
}
```

```
class Consumer implements Runnable{  
    2 usages  
    Store store;  
    1 usage  
    Consumer(Store store){this.store=store;}  
    public void run(){  
        for (int i = 1; i < 6; i++){  
            store.get();  
        }  
    }  
}
```

Взаємодія потоків. Методи wait та notify

```
class Store{
    6 usages
    private int product=0;
    1 usage
    public synchronized void get() {
        while (product<1) {
            try {
                wait();
            }
            catch (InterruptedException e) {
            }
        }
        product--;
        System.out.println("Покупець купив 1 товар");
        System.out.println("Товарів на складі: " + product);
        notify();
    }

    1 usage
    public synchronized void put() {
        while (product>=3) {
            try {
                wait();
            }
            catch (InterruptedException e) {
            }
        }
        product++;
        System.out.println("Виробник додав 1 товар");
        System.out.println("Товарів на складі: " + product);
        notify();
    }
}
```

```
class Program {
    public static void main(String[] args) {
        Store store=new Store();
        Producer producer = new Producer(store);
        Consumer consumer = new Consumer(store);
        new Thread(producer).start();
        new Thread(consumer).start();
    }
}
```

```
Виробник додав 1 товар
Товарів на складі: 1
Виробник додав 1 товар
Товарів на складі: 2
Виробник додав 1 товар
Товарів на складі: 3
Покупець купив 1 товар
Товарів на складі: 2
Покупець купив 1 товар
Товарів на складі: 1
Покупець купив 1 товар
Товарів на складі: 0
Виробник додав 1 товар
Товарів на складі: 1
Виробник додав 1 товар
Товарів на складі: 2
Покупець купив 1 товар
Товарів на складі: 1
Покупець купив 1 товар
Товарів на складі: 0
```

Взаємодія потоків. Методи `wait` та `notify`

Отже, вище визначено клас магазину, споживача і покупця. Виробник у методі `run()` додає в об'єкт **Store** за допомогою його методу `put()` 5 товарів. Споживач у методі `run()` у циклі звертається до методу `get` об'єкта **Store** для отримання цих товарів. Обидва методи **Store** - `put` і `get` є синхронізованими. Для відстеження наявності товарів у класі **Store** перевіряємо значення змінної **product**.

За замовчуванням товару немає, тому змінна дорівнює 0. Метод **get()** - отримання товару має спрацьовувати тільки за наявності хоча б одного товару. Тому в методі `get` перевіряємо, чи відсутній товар:

```
while (product<1)
```

Якщо товар відсутній, викликається метод **wait()**. Цей метод звільняє монітор об'єкта **Store** і блокує виконання методу `get`, поки для цього ж монітора не буде викликано метод **notify()**.

Взаємодія потоків. Методи `wait` та `notify`

Коли в методі `put()` додається товар і викликається `notify()`, то метод `get()` отримує монітор і виходить із конструкції `while (product<1)`, оскільки товар додано. Потім імітується отримання покупцем товару. Для цього виводиться повідомлення, і зменшується значення `product`: `product--`. І в кінці виклик методу `notify()` дає сигнал методу `put()` продовжити роботу. У методі `put()` працює схожа логіка, тільки тепер метод `put()` має спрацьовувати, якщо в магазині не більше трьох товарів. Тому в циклі перевіряється наявність товару, і якщо товар уже є, то звільняємо монітор за допомогою `wait()` і чекаємо на виклик `notify()` у методі `get()`.

Таким чином, за допомогою `wait()` у методі `get()` ми очікуємо, коли виробник додасть новий продукт. А після додавання викликаємо `notify()`, ніби кажучи, що на складі звільнилося одне місце, і можна ще додавати. А в методі `put()` за допомогою `wait()` ми очікуємо звільнення місця на складі. Після того, як місце звільниться, додаємо товар і через `notify()` повідомляємо покупця про те, що він може забирати товар.

Семафори

Семафори представляють ще один засіб синхронізації для доступу до ресурсу. У Java семафори представлені класом `Semaphore`, який розташовується в пакеті **`java.util.concurrent`**. Для управління доступом до ресурсу семафор використовує лічильник, що представляє кількість дозволів. Якщо значення лічильника більше нуля, то потік отримує доступ до ресурсу, при цьому лічильник зменшується на одиницю.

Після закінчення роботи з ресурсом потік звільняє семафор, і лічильник збільшується на одиницю. Якщо ж лічильник дорівнює нулю, то потік блокується і чекає, поки не отримає дозвіл від семафора.

Встановити кількість дозволів для доступу до ресурсу можна за допомогою конструкторів класу **`Semaphore`**:

```
Semaphore(int permits)
```

```
Semaphore(int permits, boolean fair)
```

Семафори

Параметр **permits** вказує на кількість допустимих дозволів для доступу до ресурсу. Параметр **fair** у другому конструкторі дає змогу встановити черговість отримання доступу. Якщо він дорівнює **true**, то дозволи надаватимуться потокам, які очікують, у тому порядку, в якому вони запитували доступ. Якщо ж він дорівнює **false**, то дозволи будуть надаватися в невизначеному порядку. Для отримання дозволу в семафора треба викликати метод **acquire()**, який має дві форми:

```
void acquire() throws InterruptedException
```

```
void acquire(int permits) throws InterruptedException
```

Для отримання одного дозволу застосовується перший варіант, а для отримання кількох дозволів - другий варіант. Після виклику цього методу поки потік не отримає дозвіл, він блокується. Після закінчення роботи з ресурсом отриманий раніше дозвіл треба звільнити за допомогою методу **release()**:

```
void release()
```

```
void release(int permits)
```

Семафори

Перший варіант методу звільняє один дозвіл, а другий варіант - кількість дозволів, зазначених у **permits**. Використовуємо семафор у простому прикладі:

```
class CommonResource{
    3 usages
    int x=0;
}

3 usages
class CountThread implements Runnable{
    4 usages
    CommonResource res;
    3 usages
    Semaphore sem;
    4 usages
    String name;
    3 usages
    CountThread(CommonResource res, Semaphore sem, String name){
        this.res=res;
        this.sem=sem;
        this.name=name;
    }

    public void run(){
        try{
            System.out.println(name + " очікує на дозвіл");
            sem.acquire();
            res.x=1;
            for (int i = 1; i < 5; i++){
                System.out.println(this.name + ": " + res.x);
                res.x++;
                Thread.sleep( mills: 100);
            }
        }
        catch (InterruptedException e){System.out.println(e.getMessage());}
        System.out.println(name + " звільняє дозвіл");
        sem.release();
    }
}
```

```
class Program {
    public static void main(String[] args) {
        Semaphore sem = new Semaphore( permits: 1); // 1 разрешение
        CommonResource res = new CommonResource();
        new Thread(new CountThread(res, sem, name: "CountThread 1")).start();
        new Thread(new CountThread(res, sem, name: "CountThread 2")).start();
        new Thread(new CountThread(res, sem, name: "CountThread 3")).start();
    }
}
```

Отже, тут є спільний ресурс **CommonResource** із полем **x**, яке змінюється кожним потоком. Потоки представлені класом **CountThread**, який отримує семафор і виконує деякі дії над ресурсом. В основному класі програми ці потоки запускаються.

Семафори

У підсумку ми отримаємо таке виведення в консоль:

```
CountThread 2 очікує на дозвіл
CountThread 1 очікує на дозвіл
CountThread 3 очікує на дозвіл
CountThread 2: 1
CountThread 2: 2
CountThread 2: 3
CountThread 2: 4
CountThread 2 звільняє дозвіл
CountThread 1: 1
CountThread 1: 2
CountThread 1: 3
CountThread 1: 4
CountThread 1 звільняє дозвіл
CountThread 3: 1
CountThread 3: 2
CountThread 3: 3
CountThread 3: 4
CountThread 3 звільняє дозвіл
```

Семафори чудово підходять для вирішення завдань, де треба обмежувати доступ. Наприклад, класична задача про філософів, які обідають.

Її суть: **є кілька філософів, припустімо, п'ять, але одночасно за столом можуть сидіти не більше ніж двоє. І треба, щоб усі філософи пообідали, але водночас не виникло взаємоблокування філософами один одного в боротьбі за тарілку і виделку:**

Семафори

```
class Philosopher extends Thread {
    3 usages
    Semaphore sem; // семафор, що обмежує число філософів
    // кількість прийомів їжі
    2 usages
    int num = 0;
    // умовний номер філософа
    4 usages
    int id;
    // як параметри конструктора передаємо ідентифікатор філософа і семафор
    1 usage
    Philosopher(Semaphore sem, int id) {
        this.sem = sem;
        this.id = id;
    }
    public void run() {
        try {
            while (num < 3) { // поки кількість прийомів їжі не досягне 3
                // Запитуємо у семафора дозвіл на виконання
                sem.acquire();
                System.out.println("Філософ " + id + " сідає за стіл");
                // Філософ їсть
                sleep( millis: 500);
                num++;
                System.out.println("Філософ " + id + " виходить з-за столу");
                sem.release();
                // Філософ гуляєт
                sleep( millis: 500);
            }
        } catch (InterruptedException e) {
            System.out.println("у філософа " + id + " проблеми зі здоров'ям");
        }
    }
}
```

```
class Program {
    public static void main(String[] args) {
        Semaphore sem = new Semaphore( permits: 2);
        for(int i=1;i<6;i++)
            new Philosopher(sem,i).start();
    }
}
```

У підсумку тільки два філософи зможуть одночасно перебувати за столом, а інші чекатимуть:

```
Філософ 1 сідає за стіл
Філософ 2 сідає за стіл
Філософ 2 виходить з-за столу
Філософ 1 виходить з-за столу
Філософ 3 сідає за стіл
Філософ 4 сідає за стіл
Філософ 4 виходить з-за столу
Філософ 3 виходить з-за столу
Філософ 5 сідає за стіл
Філософ 1 сідає за стіл
Філософ 1 виходить з-за столу
Філософ 5 виходить з-за столу
Філософ 2 сідає за стіл
Філософ 3 сідає за стіл
Філософ 3 виходить з-за столу
Філософ 2 виходить з-за столу
```

Обмін між потоками. Клас Exchanger

Клас **Exchanger** призначений для обміну даними між потоками. Він є типізованим і типізується типом даних, якими потоки мають обмінюватися. Обмін даними здійснюється за допомогою єдиного методу цього класу **exchange()**:

V exchange(V x) throws InterruptedException

V exchange(V x, long timeout, TimeUnit unit) throws InterruptedException, TimeoutException

Параметр **x** представляє буфер даних для обміну. Друга форма методу також визначає параметр **timeout** - час очікування і **unit** - тип часових одиниць, що застосовуються для параметра **timeout**.

Цей клас дуже просто використовувати. На наступному слайді приведено приклад використання цього класу.

Обмін між потоками. Клас Exchanger

```
class GetThread implements Runnable{
    2 usages
    Exchanger<String> exchanger;
    4 usages
    String message;
    1 usage
    GetThread(Exchanger<String> ex){
        this.exchanger=ex;
        message = "Hello World!";
    }
    public void run(){
        try{
            message=exchanger.exchange(message);
            System.out.println("GetThread has received: " + message);
        }
        catch(InterruptedException ex){
            System.out.println(ex.getMessage());
        }
    }
}
```

```
class PutThread implements Runnable{
    2 usages
    Exchanger<String> exchanger;
    4 usages
    String message;
    1 usage
    PutThread(Exchanger<String> ex){
        this.exchanger=ex;
        message = "Hello Java!";
    }
    public void run(){
        try{
            message=exchanger.exchange(message);
            System.out.println("PutThread has received: " + message);
        }
        catch(InterruptedException ex){
            System.out.println(ex.getMessage());
        }
    }
}
```

```
public static void main(String[] args) {
    Exchanger<String> ex = new Exchanger<>();
    new Thread(new PutThread(ex)).start();
    new Thread(new GetThread(ex)).start();
}
```

```
PutThread has received: Hello World!
GetThread has received: Hello Java!

Process finished with exit code 0
```

У класі **PutThread** відправляє в буфер повідомлення «Hello Java!»:
`message=exchanger.exchange(message);`

Причому у відповідь метод `exchange` повертає дані, які відправив у буфер інший потік. Тобто відбувається обмін даними. Хоча нам необов'язково отримувати дані, ми можемо просто їх відправити:

`exchanger.exchange(message);`

Логіка класу **GetThread** аналогічна – також надсилається повідомлення.

Клас Phaser

Клас **Phaser** дає змогу синхронізувати потоки, що представляють окрему фазу або стадію виконання загальної дії. **Phaser** визначає об'єкт синхронізації, який чекає, поки не завершиться певна фаза. Потім Phaser переходить до наступної стадії або фази і знову очікує її завершення. Для створення об'єкта Phaser використовується один із конструкторів:

Phaser()

Phaser(int parties)

Phaser(Phaser parent)

Phaser(Phaser parent, int parties)

Параметр **parties** вказує на кількість учасників (грубо кажучи, потоків), які повинні виконувати всі фази дії. Перший конструктор створює об'єкт **Phaser** без будь-яких учасників. Другий конструктор реєструє кількість учасників, що передається в конструктор. Третій і четвертий конструктори також встановлюють батьківський об'єкт **Phaser**.

Клас Phaser

Основні методи класу **Phaser**:

- **int register()**: реєструє учасника, який виконує фази, і повертає номер поточної фази - зазвичай фаза 0;
- **int arrive()**: повідомляє, що учасник завершив фазу, і повертає номер поточної фази;
- **int arriveAndAwaitAdvance()**: аналогічний методу arrive, тільки при цьому змушує phaser очікувати на завершення фази всіма іншими учасниками;
- **int arriveAndDeregister()**: повідомляє про завершення всіх фаз учасником і знімає його з реєстрації. Повертає номер поточної фази або від'ємне число, якщо синхронізатор Phaser завершив свою роботу;
- **int getPhase()**: повертає номер поточної фази.

Під час роботи з класом **Phaser** зазвичай спочатку створюється його об'єкт. Далі нам треба зареєструвати всіх учасників. Для реєстрації для кожного учасника викликається метод **register()**, або можна обійтися і без цього методу, передавши потрібну кількість учасників у конструктор **Phaser**. Потім кожен учасник виконує деякий набір дій, що складають фазу. А синхронізатор Phaser чекає, поки всі учасники не завершать виконання фази. Щоб повідомити синхронізатору, що фаза завершена, учасник має викликати метод **arrive()** або **arriveAndAwaitAdvance()**. Після цього синхронізатор переходить до наступної фази.

Клас Phaser

Застосуємо **Phaser** у застосунку:

```
class PhaseThread implements Runnable{
    8 usages
    Phaser phaser;
    4 usages
    String name;
    2 usages
    PhaseThread(Phaser p, String n){
        this.phaser=p;
        this.name=n;
        phaser.register();
    }
    public void run(){
        System.out.println(name + " виконує фазу " + phaser.getPhase());
        phaser.arriveAndAwaitAdvance(); // повідомляємо, що першу фазу досягнуто
        System.out.println(name + " виконує фазу " + phaser.getPhase());
        phaser.arriveAndAwaitAdvance(); // повідомляємо, що другу фазу досягнуто
        System.out.println(name + " виконує фазу " + phaser.getPhase());
        phaser.arriveAndDeregister(); // повідомляємо про завершення фаз і видаляємо з реєстрації об'єкти
    }
}
```

```
public static void main(String[] args) {
    Phaser phaser = new Phaser( parties: 1);
    new Thread(new PhaseThread(phaser, n: "PhaseThread 1")).start();
    new Thread(new PhaseThread(phaser, n: "PhaseThread 2")).start();
    // чекаємо завершення фази 0
    int phase = phaser.getPhase();
    phaser.arriveAndAwaitAdvance();
    System.out.println("Фазу " + phase + " завершено");
    // чекаємо завершення фази 1
    phase = phaser.getPhase();
    phaser.arriveAndAwaitAdvance();
    System.out.println("Фазу " + phase + " завершено");
    // чекаємо завершення фази 2
    phase = phaser.getPhase();
    phaser.arriveAndAwaitAdvance();
    System.out.println("Фазу " + phase + " завершено");

    phaser.arriveAndDeregister();
}
```

Отже, тут у нас фази виконуються трьома учасниками - головним потоком і двома потоками **PhaseThread**. Тому під час створення об'єкта **Phaser** йому передається число 1 - головний потік, а в конструкторі **PhaseThread** викликається метод `register()`. Ми в принципі могли б не використовувати метод `register`, але тоді нам треба було б вказати **Phaser phaser = new Phaser(3)**, оскільки у нас три учасники.

Клас Phaser

Фаза для кожного учасника являє собою мінімальний примітивний набір дій: для потоків **PhaseThread** це виведення повідомлення, а для головного потоку - підрахунок поточної фази за допомогою методу **getPhase()**. При цьому відлік фаз починається з нуля. Кожен учасник завершує виконання фази викликом методу **phaser.arriveAndAwaitAdvance()**. Під час виклику цього методу доки останній учасник не завершить виконання поточної фази, всі інші учасники блокуються. Після завершення виконання останньої фази відбувається скасування реєстрації всіх учасників за допомогою методу **arriveAndDeregister()**.

У підсумку робота програми дасть такий висновок:

```
PhaseThread 2 виконує фазу 0
PhaseThread 1 виконує фазу 0
PhaseThread 1 виконує фазу 1
PhaseThread 2 виконує фазу 1
Фаза 0 завершено
Фаза 1 завершено
PhaseThread 2 виконує фазу 2
PhaseThread 1 виконує фазу 2
Фаза 2 завершено

Process finished with exit code 0
```

У цьому випадку виходить трохи **плутаний висновок**. Так, повідомлення про виконання фази 1 виводиться після повідомлення про закінчення фази 0. Що пов'язано з багатопотоковістю - фази завершилися, але в одному потоці ще не виведено повідомлення про завершення, тоді як інші потоки вже почали виконання наступної фази. У будь-якому разі все це відбувається вже після завершення фази.

Блокування. ReentrantLock

Для управління доступом до спільного ресурсу як альтернативу оператору **synchronized** ми можемо використовувати блокування. Функціональність блокувань міститься в пакеті **java.util.concurrent.locks**. Спочатку потік намагається отримати доступ до загального ресурсу. Якщо він вільний, то на нього накладає блокування. Після завершення роботи блокування зі спільного ресурсу знімається. Якщо ж ресурс не вільний і на нього вже накладено блокування, то потік очікує, поки це блокування не буде знято.

Класи блокувань реалізують інтерфейс **Lock**, який визначає такі методи:

- **void lock():** очікує, поки не буде отримано блокування;
- **void lockInterruptibly() throws InterruptedException:** очікує, поки не буде отримано блокування, якщо потік не перервано;
- **boolean tryLock():** намагається отримати блокування, якщо блокування отримано, то повертає true. Якщо блокування не отримано, то повертає false. На відміну від методу lock() не очікує отримання блокування, якщо воно недоступне;
- **void unlock():** знімає блокування;
- **Condition newCondition():** повертає об'єкт Condition, який пов'язаний з поточним блокуванням.

Організація блокування в загальному випадку досить проста: для отримання блокування викликається метод **lock()**, а після закінчення роботи зі спільними ресурсами викликається метод **unlock()**, який знімає блокування.

Блокування. ReentrantLock

Об'єкт **Condition** дає змогу керувати блокуванням. Як правило, для роботи з блокуваннями використовується клас **ReentrantLock** з пакета **java.util.concurrent.locks**. Цей клас реалізує інтерфейс **Lock**. Для прикладу візьмемо код з теми про оператор **synchronized** і перепишемо цей код з використанням заглушки **ReentrantLock**:

```
class CommonResource{
    3 usages
    int x=0;
}

1 usage
class CountThread implements Runnable{
    4 usages
    CommonResource res;
    3 usages
    ReentrantLock locker;
    1 usage
    CountThread(CommonResource res, ReentrantLock lock){
        this.res=res;
        locker = lock;
    }
    public void run(){
        locker.lock(); // встановлюємо блокування
        try{
            res.x=1;
            for (int i = 1; i < 5; i++){
                System.out.printf("%s %d \n", Thread.currentThread().getName(), res.x);
                res.x++;
                Thread.sleep(100);
            }
        }
        catch (InterruptedException e){
            System.out.println(e.getMessage());
        }
        finally{
            locker.unlock(); // знімаємо блокування
        }
    }
}
```

```
public static void main(String[] args) {
    CommonResource commonResource= new CommonResource();
    ReentrantLock locker = new ReentrantLock(); // створюємо заглушку
    for (int i = 1; i < 6; i++){
        Thread t = new Thread(new CountThread(commonResource, locker));
        t.setName("Thread "+ i);
        t.start();
    }
}
```

Блокування. ReentrantLock

Тут також використовується загальний ресурс **CommonResource**, для управління яким створюється п'ять потоків. На вході в критичну секцію встановлюється заглушка:

```
locker.lock();
```

Після цього тільки один потік має доступ до критичної секції, а інші потоки очікують зняття блокування. У блоці `finally` після закінчення основної роботи потоку це блокування знімається. Причому робиться це обов'язково в блоці `finally`, оскільки в разі виникнення помилки всі інші потоки виявляться заблокованими. У підсумку ми отримаємо висновок, аналогічний тому, який був у випадку з оператором ***synchronized***:

```
Thread 1 1
Thread 1 2
Thread 1 3
Thread 1 4
Thread 2 1
Thread 2 2
Thread 2 3
Thread 2 4
Thread 3 1
Thread 3 2
Thread 3 3
Thread 3 4
Thread 4 1
Thread 4 2
Thread 4 3
Thread 4 4
Thread 5 1
Thread 5 2
Thread 5 3
Thread 5 4
```

Умови в блокуваннях

Застосування умов у блокуваннях дає змогу домогтися контролю над керуванням доступом до потоків. Умова блокування являє собою об'єкт інтерфейсу **Condition** з пакета ***java.util.concurrent.locks***.

Ми можемо використовувати такі методи інтерфейсу **Condition**:

- **await**: потік очікує, поки не буде виконано деяку умову і поки інший потік не викличе методи **signal/signalAll**. Багато в чому аналогічний методу **wait** класу **Object**;
- **signal**: сигналізує, що потік, у якого раніше був викликаний метод **await()**, може продовжити роботу. Застосування аналогічне використанню методу **notify** класу **Object**;
- **signalAll**: сигналізує всім потокам, у яких раніше був викликаний метод **await()**, що вони можуть продовжити роботу. Аналогічний методу **notifyAll()** класу **Object**.

Умови в блокуваннях

Ці методи викликаються з блоку коду, який потрапляє під дію блокування **ReentrantLock**. Спочатку, використовуючи це блокування, нам треба отримати об'єкт **Condition**:

```
ReentrantLock locker = new ReentrantLock();  
Condition condition = locker.newCondition();
```

Як правило, спочатку перевіряється умова доступу. Якщо дотримується умова, то потік очікує, поки умова не зміниться:

```
while (умова)  
    condition.await();
```

Після виконання всіх дій іншим потокам подається сигнал про зміну умови:

```
condition.signalAll();
```

Важливо наприкінці викликати метод `signal/signalAll`, щоб уникнути можливості взаємоблокування потоків.

Умови в блокуваннях

Для прикладу візьмемо задачу з теми про методи **wait/notify** і змінимо її, застосовуючи об'єкт **Condition**. Отже, у нас є склад, де може одночасно бути розміщено не більше 3 товарів. І виробник має виробити 5 товарів, а покупець має ці товари купити. Водночас покупець не може купити товар, якщо на складі немає жодних товарів:

```
class Producer implements Runnable{
    2 usages
    Store store;

    1 usage
    Producer(Store store){this.store=store;}

    public void run(){
        for (int i = 1; i < 6; i++){
            store.put();
        }
    }
}

2 usages
class Consumer implements Runnable{
    2 usages
    Store store;

    1 usage
    Consumer(Store store){this.store=store;}

    public void run(){
        for (int i = 1; i < 6; i++){
            store.get();
        }
    }
}
```

```
class Store{
    6 usages
    private int product=0;
    6 usages
    ReentrantLock locker;
    5 usages
    Condition condition;
    1 usage
    Store(){
        locker = new ReentrantLock(); // створимо блокування
        condition = locker.newCondition(); // отримуємо умову, пов'язану з блокуванням
    }

    1 usage
    public void get() {
        locker.lock();
        try{
            // поки немає доступних товарів на складі, очікуємо
            while (product<1)
                condition.await();
            product--;
            System.out.println("Покупець купив 1 товар");
            System.out.println("Товарів на складі: " + product);

            // сигналізуємо
            condition.signalAll();
        }
        catch (InterruptedException e){
            System.out.println(e.getMessage());
        }
        finally{
            locker.unlock();
        }
    }
}
```

```
public void put() {
    locker.lock();
    try{
        // поки що на складі 3 товари, чекаємо звільнення місця
        while (product>=3)
            condition.await();

        product++;
        System.out.println("Виробник додав 1 товар");
        System.out.println("Товарів на складі: " + product);
        // сигналізуємо
        condition.signalAll();
    }
    catch (InterruptedException e){
        System.out.println(e.getMessage());
    }
    finally{
        locker.unlock();
    }
}
```

Умови в блокуванні

```
class Program {  
    public static void main(String[] args) {  
        Store store=new Store();  
        Producer producer = new Producer(store);  
        Consumer consumer = new Consumer(store);  
        new Thread(producer).start();  
        new Thread(consumer).start();  
    }  
}
```

У підсумку ми отримаємо виведення в консоль:

```
Виробник додав 1 товар  
Товарів на складі: 1  
Виробник додав 1 товар  
Товарів на складі: 2  
Виробник додав 1 товар  
Товарів на складі: 3  
Покупець купив 1 товар  
Товарів на складі: 2  
Покупець купив 1 товар  
Товарів на складі: 1  
Покупець купив 1 товар  
Товарів на складі: 0  
Виробник додав 1 товар  
Товарів на складі: 1  
Виробник додав 1 товар  
Товарів на складі: 2  
Покупець купив 1 товар  
Товарів на складі: 1  
Покупець купив 1 товар  
Товарів на складі: 0  
  
Process finished with exit code 0
```

Контрольні питання

1. Які переваги забезпечує багатопотокове програмування в мові програмування Java?
2. Як можна створити потік у мові програмування Java, і які існують способи для цього?
3. В чому відмінності між класами Thread і Runnable у контексті багатопоточності в мові програмування Java?
4. Що означає термін "синхронізація" і чому вона важлива в багатопотокових додатках?
5. Що представляє собою монітор, і як він пов'язаний із синхронізацією в мові програмування Java?
6. Як працює ключове слово synchronized, і в яких випадках його слід використовувати?
7. Які недоліки має багатопотокове програмування в мові програмування Java?

The END
Lec 12.

.NET та JAVA #13

Робота з БД. Ознайомлення з JDBC

1. Введення в бази даних
2. SQL
3. Java та бази даних. JDBC
4. Відмінність між Statement та PreparedStatement

Введення в бази даних

База даних – це організований набір структурованої інформації або даних, які зазвичай зберігаються в електронному вигляді в комп'ютерній системі. Зазвичай базою даних керує система управління базами даних (**СУБД**). Разом дані та **СУБД**, а також додатки, які з ними пов'язані, називаються системою баз даних, яку часто скорочують до просто бази даних.

Дані в найпоширеніших типах баз даних, що використовуються сьогодні, зазвичай моделюються у вигляді рядків і стовпців у серії таблиць, щоб зробити обробку та запити до даних ефективними. Тоді до даних можна легко отримати доступ, керувати ними, змінювати, оновлювати, контролювати та організовувати. Більшість баз даних використовують мову структурованих запитів (**SQL**) для запису і запитів до даних.

Система управління базами даних - СУБД (Database Management System – DBMS) – це технологія зберігання та пошуку даних користувачів з максимальною ефективністю та відповідними заходами безпеки. Приклади сучасних СУБД: **PostgreSQL, MySQL, Oracle, MongoDB**.

Традиційно дані були організовані у файлових форматах. Тоді СУБД була новою концепцією, і всі дослідження були спрямовані на те, щоб зробити її здатною подолати недоліки традиційного стилю управління даними. Сучасна СУБД має наступні основні характеристики:

- Використання цифрового сховища, створеного на сервері, для зберігання та управління інформацією;
- Забезпечення чіткого і логічного уявлення про процес, який маніпулює даними;

Введення в бази даних

- СУБД містить автоматичні процедури резервного копіювання та відновлення;
- Містить властивості **ACID** (***atomicity, consistency, isolation, durability***), які підтримують дані у здоровому стані у випадку збою. Це може зменшити складні взаємозв'язки між даними;
- Підтримка маніпулювання та обробки даних;
- Забезпечення безпеки даних;
- Переглядання БД з різних точок зору відповідно до вимог користувача.

SQL

SQL - це стандартна мова баз даних, яка використовується для доступу та маніпулювання даними в базах даних. SQL розшифровується як Structured Query Language (мова структурованих запитів). SQL була розроблена фахівцями IBM Computer Scientists у 1970-х роках.

Виконуючи запити, SQL може створювати, оновлювати, видаляти та отримувати дані в таких базах даних, як MySQL, Oracle, PostgreSQL тощо. Загалом SQL - це мова запитів, яка взаємодіє з базами даних.

Даний курс не включає в себе вивчення мови SQL, але вона не складна для самостійного вивчення!

Java та бази даних. JDBC

Для зберігання даних ми можемо використовувати різні бази даних - **Oracle, MS SQL Server, MySQL, PostgreSQL** тощо. Усі ці системи управління базами даних мають свої особливості. Головне, що їх об'єднує, це взаємодія зі сховищем даних за допомогою команд SQL. І щоб визначити єдиний механізм взаємодії з цими СУБД у Java ще починаючи з 1996 року було введено спеціальний прикладний інтерфейс API, який називається **JDBC**.

Тобто якщо ми хочемо в додатку мовою Java взаємодіяти з базою даних, то необхідно використовувати функціональні можливості JDBC. Даний API входить до складу Java, зокрема, для роботи з JDBC в програмі Java досить підключити пакет **java.sql**. Для роботи в **Java EE** є аналогічний пакет **javax.sql**, який розширює можливості JDBC.

Java та бази даних. JDBC

JDBC розшифровується як **Java Database Connectivity**, що є стандартним Java API для незалежного від бази даних зв'язку між мовою програмування Java та широким спектром баз даних. Бібліотека JDBC містить API для кожної з перелічених нижче задач, які зазвичай пов'язані з використанням баз даних:

- Встановлення з'єднання з базою даних;
- Створення операторів SQL або MySQL;
- Виконання SQL або MySQL запитів до бази даних;
- Перегляд та модифікація отриманих записів.

По суті, **JDBC** – це специфікація, яка надає повний набір інтерфейсів, що забезпечує портативний доступ до баз даних. Java можна використовувати для написання різних типів виконуваних файлів, таких як: **Java додатки, Java-аплети, сервлети Java, Java ServerPages (JSP), Enterprise JavaBeans (EJB)**.

Всі ці різні виконувані програми можуть використовувати драйвер **JDBC** для доступу до бази даних і користуватися перевагами збережених даних. **JDBC** надає ті ж можливості, що і **ODBC**, дозволяючи Java-програмам містити незалежний від бази даних код.

Java та бази даних. JDBC

Архітектура JDBC

JDBC API підтримує як дворівневі, так і трирівневі моделі обробки для доступу до баз даних, але в цілому архітектура JDBC складається з двох рівнів:

- **JDBC API** - забезпечує з'єднання програми з JDBC Manager;
- **JDBC Driver API** - підтримує з'єднання JDBC Manager з драйвером.

JDBC API використовує менеджер драйверів і драйвери для конкретних баз даних, щоб забезпечити прозоре підключення до різнорідних баз даних.

Менеджер драйверів JDBC гарантує, що для доступу до кожного джерела даних використовується правильний драйвер. Диспетчер драйверів може підтримувати декілька паралельних драйверів, підключених до декількох гетерогенних баз даних.

Загальні компоненти JDBC

JDBC API надає наступні інтерфейси та класи:

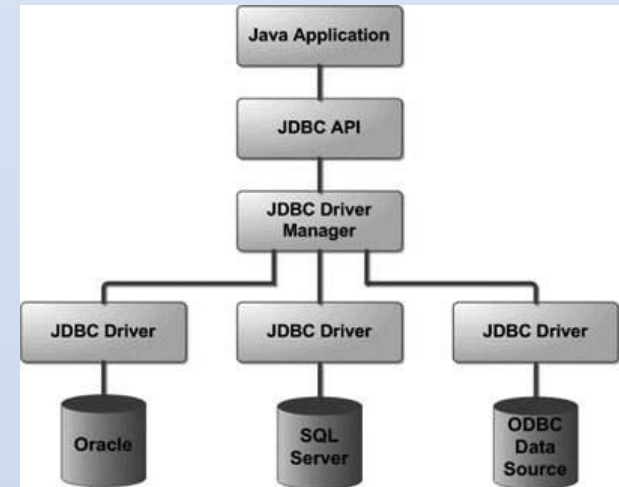
- **DriverManager** - цей клас керує списком драйверів баз даних. Зіставляє запити на з'єднання від java-додатку з відповідним драйвером бази даних, використовуючи підпротокол зв'язку. Перший драйвер, який розпізнає певний підпротокол під JDBC, буде використаний для встановлення з'єднання з базою даних.

Java та бази даних. JDBC

- **Driver** - цей інтерфейс відповідає за зв'язок з сервером бази даних. Ви будете взаємодіяти безпосередньо з об'єктами **Driver** дуже рідко. Замість цього ви використовуєте об'єкти **DriverManager**, який керує об'єктами цього типу. Він також абстрагує деталі, пов'язані з роботою з об'єктами Driver.
- **Connection** - це інтерфейс з усіма методами для зв'язку з базою даних. Об'єкт з'єднання представляє контекст зв'язку, тобто весь зв'язок з базою даних відбувається тільки через об'єкт з'єднання.
- **Statement** – це інтерфейс, який необхідний для надсилання **SQL-запитів** до бази даних. Деякі похідні інтерфейси приймають параметри на додаток до виконання збережених процедур.
- **ResultSet** - інтерфейс, об'єкти якого необхідні для отримання даних з бази даних після виконання SQL-запиту за допомогою об'єктів Statement. Він діє як ітератор, щоб дозволити вам переміщатися по його даних.
- **SQLException** - цей клас обробляє будь-які помилки, що виникають у додатку бази даних.

Java та бази даних. JDBC

Не всі бази даних можуть підтримуватися через **JDBC**. Для роботи з певною **СУБД** також необхідний спеціальний драйвер. Кожен розробник певної **СУБД** зазвичай надає свій драйвер для роботи з **JDBC**. Тобто якщо ми хочемо працювати з **MySQL**, то нам потрібен спеціальний драйвер для роботи саме **MySQL**. Як правило, більшість драйверів доступні у вільному доступі на сайтах відповідних СУБД. Зазвичай вони представляють **JAR-файли**. І перевага **JDBC** якраз і полягає в тому, що ми абстрагуємося від будови конкретної бази даних, а використовуємо уніфікований інтерфейс, який єдиний для всіх.



Java та бази даних. JDBC

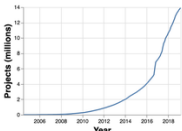
Для встановлення драйвера ми можемо використати **Maven** - фреймворк для автоматизації складання проєктів. Для цього переходимо за посиланням <https://mvnrepository.com/> та в пошуку знаходимо драйвер **PostgreSQL** та копіюємо залежність знизу сторінки. (Якщо ми хочемо використати іншу СУБД, то ми шукаємо інший драйвер, наприклад **MySQL**)

MVN REPOSITORY

Search for groups, artifacts, categories

Search

Indexed Artifacts (35.4M)



Popular Categories

Testing Frameworks & Tools

Android Packages

Logging Frameworks

Java Specifications

JSON Libraries

JVM Languages

Language Runtime

Core Utilities

Mocking

Web Assets

Annotation Libraries

Logging Bridges

HTTP Clients

Dependency Injection

Home » org.postgresql » postgresql » 42.6.0

 **PostgreSQL JDBC Driver » 42.6.0**
PostgreSQL JDBC Driver Postgresql

License	BSD 2-clause
Categories	JDBC Drivers
Tags	database sql jdbc postgresql driver
Organization	PostgreSQL Global Development Group
HomePage	https://jdbc.postgresql.org
Date	Mar 18, 2023
Files	pom (2 KB) jar (1.0 MB) View All
Repositories	Central
Ranking	#115 in MvnRepository (See Top Artifacts) #2 in JDBC Drivers
Used By	4,119 artifacts

Maven

Gradle

Gradle (Short)

Gradle (Kotlin)

SBT

Ivy

Grape

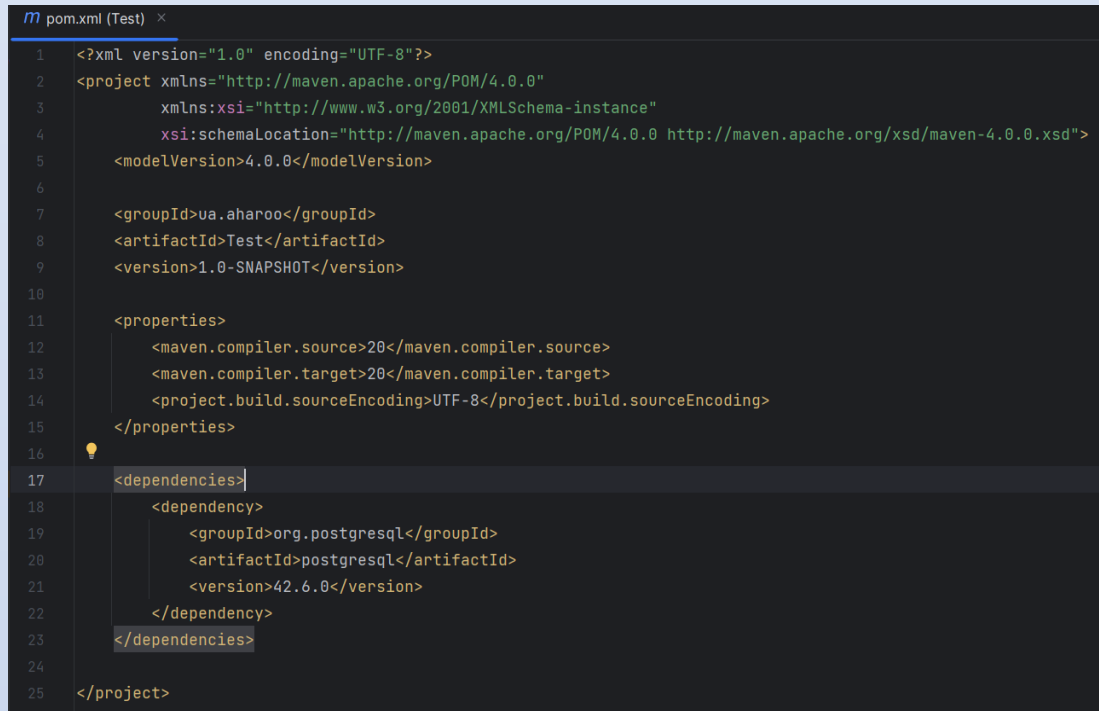
Leiningen

Buildr

```
<!-- https://mvnrepository.com/artifact/org.postgresql/postgresql -->
<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <version>42.6.0</version>
</dependency>
```

Java та бази даних. JDBC

Тепер створимо проект в IntelliJ IDEA, після цього переходимо до файлу **pom.xml**, та вставляємо залежність.



```
m pom.xml (Test) x
1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
5      <modelVersion>4.0.0</modelVersion>
6
7      <groupId>ua.aharoo</groupId>
8      <artifactId>Test</artifactId>
9      <version>1.0-SNAPSHOT</version>
10
11     <properties>
12         <maven.compiler.source>20</maven.compiler.source>
13         <maven.compiler.target>20</maven.compiler.target>
14         <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
15     </properties>
16
17     <dependencies>
18         <dependency>
19             <groupId>org.postgresql</groupId>
20             <artifactId>postgresql</artifactId>
21             <version>42.6.0</version>
22         </dependency>
23     </dependencies>
24
25 </project>
```

Переконаємося, що ми можемо здійснювати взаємодію з **PostgreSQL** через цей драйвер. Для цього визначимо такий код програми:

Java та бази даних. JDBC

```
class Test{
    public static void main(String[] args) {
        try{
            Class.forName("org.postgresql.Driver").getDeclaredConstructor().newInstance();
            System.out.println("Connection successful!");
        }
        catch(Exception ex){
            System.out.println("Connection failed!");
            System.out.println(ex);
        }
    }
}
```

```
Connection successful!
```

```
Process finished with exit code 0
```

Для завантаження драйвера застосовується рядок:

```
Class.forName("org.postgresql.Driver").getDeclaredConstructor().newInstance();
```

Метод **Class.forName()** як параметр приймає рядок, який представляє повний шлях до класу драйвера з урахуванням усіх пакетів. У випадку PostgreSQL це шлях **"org.postgresql.Driver"**. Таким чином, Метод **Class.forName** завантажує клас драйвера, який буде використовуватися.

Далі викликається метод **getDeclaredConstructor()**, який повертає конструктор даного класу. І в кінці викликається метод **newInstance()**, який створює за допомогою конструктора об'єкт даного класу. І після цього ми зможемо взаємодіяти з сервером **PostgreSQL**.

Java та бази даних. JDBC

Підключення до бази даних

Для підключення до бази даних необхідно створити об'єкт **java.sql.Connection**. Для його створення застосовується метод:

Connection DriverManager.getConnection(url, username, password)

Метод **DriverManager.getConnection** як параметри приймає адресу джерела даних, логін і пароль. Як логін і пароль передаються логін і пароль від сервера PostgreSQL. Адреса локальної бази даних PostgreSQL вказується в такому форматі: ***jdbc:postgresql://localhost/назва_бази_даних***

Приклад створення підключення до створеної вище локальної бази даних store:

```
try
    (Connection
    DriverManager.getConnection("jdbc:postgresql://localhost/store",
    "password")){}
    conn
    "root",
    =
```

Java та бази даних. JDBC

Файли конфігурації

Ми можемо визначити всі дані для підключення безпосередньо в програмі. Однак що якщо якісь дані було змінено? У цьому разі знадобиться перекомпіляція програми. Іноді це не завжди зручно, наприклад, відсутній доступ до вихідних кодів, або перекомпіляція займе досить тривалий час. У цьому випадку ми можемо зберігати налаштування у файлі.

Так, створимо в папці програми новий текстовий файл **database.properties**, у якому визначимо налаштування підключення:

```
url = "jdbc:postgresql://localhost:5432/test";username = "postgres";password = "password";
```

Завантажимо ці налаштування в програмі:

```
class Test{
    public static void main(String[] args) {
        try {
            Class.forName("org.postgresql.Driver").getDeclaredConstructor().newInstance();
            try (Connection conn = getConnection()) {
                System.out.println("Connection to Store DB succesfull!");
            }
        } catch (Exception ex) {
            System.out.println("Connection failed...");
            System.out.println(ex);
        }
    }
}

1 usage
public static Connection getConnection() throws SQLException, IOException {
    Properties props = new Properties();
    try (InputStream in = Files.newInputStream(Paths.get("D:/Study/Test1/src/main/java/database.properties"))) {
        props.load(in);
    }
    String url = props.getProperty("url");
    String username = props.getProperty("username");
    String password = props.getProperty("password");

    return DriverManager.getConnection(url, username, password);
}
```

Java та бази даних. JDBC

Виконання команд. Метод `executeUpdate`

Для взаємодії з базою даних додаток надсилає серверу PostgreSQL команди мовою SQL. Щоб виконати команду, спочатку необхідно створити об'єкт `Statement`.

Для його створення в об'єкта `Connection` викликається метод `createStatement()`:

```
Statement statement = conn.createStatement();
```

Для виконання команд SQL у класі `Statement` визначено три методи:

- **`executeUpdate`**: виконує такі команди, як **`INSERT`**, **`UPDATE`**, **`DELETE`**, **`CREATE TABLE`**, **`DROP TABLE`**. Як результат повертає кількість рядків, зачеплених операцією (наприклад, кількість доданих, змінених або видалених рядків), або 0, якщо жодного рядка не зачеплено операцією або якщо команда не змінює вміст таблиці (наприклад, команда створення нової таблиці);
- **`executeQuery`**: виконує команду **`SELECT`**. Повертає об'єкт **`ResultSet`**, який містить результати запиту;
- **`execute()`**: виконує будь-які команди і повертає значення `boolean: true` - якщо команда повертає набір рядків (**`SELECT`**), інакше повертається `false`.

Java та бази даних. JDBC

Метод `executeQuery`. Отримання даних

Для вибірки даних за допомогою команди `SELECT` застосовується метод `executeQuery`:

```
ResultSet executeQuery("Команда_SQL")
```

Метод повертає об'єкт **ResultSet**, який містить усі отримані дані. В об'єкті **ResultSet** ітератор встановлюється на позиції перед першим рядком. І щоб переміститися до першого рядка (і до всіх наступних) необхідно викликати метод `next()`.

Поки в наборі **ResultSet** є доступні рядки, метод `next` повертатиме `true`. Типове переміщення по набору рядків:

```
ResultSet resultSet = statement.executeQuery("SELECT * FROM Products");  
while(resultSet.next()){// отримання вмісту рядків}
```

Тобто поки в `resultSet` є доступні рядки, виконуватиметься цикл `while`, який переходить до наступного рядка в наборі.

Після переходу до рядка ми можемо отримати його вміст. Для цього у `ResultSet` визначено низку методів. Деякі з них:

- **`getBoolean()`** повертає значення `boolean`;
- **`getDate()`** повертає значення `Date`;
- **`getDouble()`** повертає значення `double`;
- **`getInt()`** повертає значення `int`;
- **`getFloat()`** повертає значення `float`;
- **`getLong()`** повертає значення `long`;
- **`getString()`** повертає значення `String`;
- **`getNString()`** повертає значення `String`;

Java та бази даних. JDBC

Залежно від того, дані якого типу зберігаються в тому чи іншому стовпці, ми можемо використовувати той чи інший метод. Кожен із цих методів має дві версії:

```
int getInt(int columnIndex)
int getInt(String columnLabel)
```

Перша версія отримує дані зі стовпця з номером **columnIndex**. Друга версія отримує дані зі стовпця з назвою **columnLabel**.

PreparedStatement

Крім класу Statement у **java.sql** ми можемо використовувати для виконання запитів ще один клас - **PreparedStatement**. Крім власне виконання запиту цей клас дозволяє підготувати запит, відформатувати його належним чином.

Наприклад, створимо таблицю яка має три стовпця:

```
CREATE TABLE products (
    Id SERIAL PRIMARY KEY,
    ProductName VARCHAR(20),
    Price INT
)
```

За допомогою PreparedStatement додамо в неї один об'єкт:

Java та бази даних. JDBC

```
class Test {  
    public static void main(String[] args) {  
        try {  
            Scanner scanner = new Scanner(System.in);  
            Class.forName("org.postgresql.Driver").getDeclaredConstructor().newInstance();  
            System.out.println("Enter product name:");  
            String name = scanner.nextLine();  
  
            System.out.println("Enter product price:");  
            int price = scanner.nextInt();  
  
            try (Connection conn = getConnection()) {  
                String sql = "INSERT INTO Products (ProductName, Price) Values (?, ?)";  
                PreparedStatement preparedStatement = conn.prepareStatement(sql);  
                preparedStatement.setString(1, name);  
                preparedStatement.setInt(2, price);  
  
                int rows = preparedStatement.executeUpdate();  
  
                System.out.printf("%d rows added", rows);  
            }  
        } catch (Exception ex) {  
            System.out.println("Connection failed...");  
  
            System.out.println(ex);  
        }  
    }  
}
```

Java та бази даних. JDBC

У цьому разі дані вводяться з консолі і потім додаються в базу даних. Для створення об'єкта **PreparedStatement** застосовується метод **prepareStatement()** інтерфейсу **Connection**. У цей метод передається вираз sql **INSERT INTO Products (ProductName, Price) Values (?, ?)**. Цей вираз може містити знаки запитання ? - знаки підстановки, замість яких будуть вставлятися реальні значення.

Щоб пов'язати окремі знаки підстановки з конкретними значеннями, у інтерфейсі **PreparedStatement** визначено низку методів для різних типів даних. Усі методи, які постачають значення замість знаків підстановки, як перший параметр приймають порядковий номер знака підстановки (нумерація починається з 1), а як другий параметр - власне значення, яке вставляється замість знака підстановки.

Наприклад, перший знак підстановки ? у виразі sql представляє значення для стовпця ProductName, який зберігає рядок. Тому для зв'язку значення з першим знаком підстановки застосовується метод **preparedStatement.setString(1, name)**.

Другий знак підстановки повинен передавати значення для стовпця Price, який зберігає цілі числа. Тому для вставки значення використовується метод **preparedStatement.setInt(2, price)**

Крім **setString** і **setInt** **PreparedStatement** має ще низку подібних методів, які працюють подібним чином. Деякі з них: **setBigDecimal**, **setBoolean**, **setDate**, **setDouble**, **setFloat**, **setLong**, **setNull**, **setTime**.

Java та бази даних. JDBC

Для виконання запиту **PreparedStatement** має три методи:

- **boolean execute()**: виконує будь-яку SQL-команду;
- **ResultSet executeQuery()**: виконує команду **SELECT**, яка повертає дані у вигляді **ResultSet**;
- **int executeUpdate()**: виконує такі **SQL-команди**, як **INSERT**, **UPDATE**, **DELETE**, **CREATE** і повертає кількість змінених рядків.

При цьому на відміну від методів **Statement** ці методи не приймають SQL-вираз.

Відмінність між Statement та PreparedStatement

Statement та **PreparedStatement** можна використовувати для виконання SQL-запитів. Ці інтерфейси виглядають дуже схожими. Однак вони суттєво відрізняються один від одного за можливостями та продуктивністю:

- **Statement** - використовується для виконання рядкових SQL-запитів;
- **PreparedStatement** - використовується для виконання параметризованих SQL-запитів.

Statement

По-перше код, написаний з використанням інтерфейсу **Statement** стає менш читабельним при конкатенації **SQL-рядків**. По-друге, він вразливий до **SQL-ін'єкцій**. Наступні приклади ілюструють цю слабкість. Створимо клас **Person**:

```
class Person {  
    no usages  
    private int id;  
    no usages  
    private String name;  
  
    // стандартні getters та setters  
}
```

```
public void insert(Person person, Connection connection) throws SQLException {  
    String query = "INSERT INTO persons(id, name) VALUES(" + person.getId() + ","  
        + person.getName() + ")";  
  
    Statement statement = connection.createStatement();  
    statement.executeUpdate(query);  
}
```

Відмінність між Statement та PreparedStatement

По-третє, **JDBC** передає запит з вбудованими значеннями до бази даних. Таким чином, немає ніякої оптимізації запиту, і, найголовніше, рушій бази даних повинен забезпечити всі перевірки. Крім того, запит не буде відображатися в базі даних як однаковий, і це перешкоджатиме використанню кешу. Аналогічно, пакетні оновлення потрібно виконувати окремо:

```
public void insert(List<Person> persons, Connection connection) throws SQLException {  
    for (Person person: persons) {  
        insert(person, connection);  
    }  
}
```

PreparedStatement

По-перше, **PreparedStatement** розширює інтерфейс **Statement**. Він має методи для зв'язування різних типів об'єктів, включаючи файли та масиви. Таким чином, код стає більш зрозумілим:

Відмінність між Statement та PreparedStatement

```
public void insert(Person personEntity, Connection connection) throws SQLException {  
    String query = "INSERT INTO persons(id, name) VALUES( ?, ?)";  
  
    PreparedStatement preparedStatement = connection.prepareStatement(query);  
    preparedStatement.setInt( parameterIndex: 1, personEntity.getId());  
    preparedStatement.setString( parameterIndex: 2, personEntity.getName());  
    preparedStatement.executeUpdate();  
}
```

По-друге, він захищає від **SQL-ін'єкцій**, екрануючи текст для всіх наданих значень параметрів.

По-третє, **PreparedStatement** використовує прекомпіляцію. Як тільки база даних отримує запит, вона перевіряє кеш, перш ніж прекомпілювати запит.

Отже, якщо він не буде закешований, рушій бази даних збереже його для наступного використання. Крім того, ця функція прискорює зв'язок між базою даних і JVM за допомогою двійкового протоколу, що не є SQL. Це означає, що в пакетах менше даних, тому зв'язок між серверами відбувається швидше.

Контрольні питання

1. Як визначається термін "база даних"?
2. Якова природа системи управління базами даних?
3. Які конкретні системи управління базами даних вам відомі?
4. Як ви визначаєте технологію JDBC?
5. Розкажіть про алгоритм підключення до СУБД використовуючи мову Java.
6. Охарактеризуйте клас PreparedStatement. Яка відмінність PreparedStatement від Statement.
7. Яким чином можна виконати SQL запити в Java?

The END
Lec 13.

.NET та JAVA #14

Основи тестування в Java

1. Введення в тестування програмного забезпечення
2. Типи тестування програмного забезпечення
3. Рівні тестування програмного забезпечення
4. Модульне тестування. Фреймворк JUnit
5. Анотації в JUnit
6. Приклад роботи Junit
7. Test-driven development

Введення в тестування ПЗ

Тестування програмного забезпечення (Software testing) - це метод оцінки функціональності програмного забезпечення. Процес перевіряє, чи відповідає фактичне програмне забезпечення очікуваним вимогам і чи не містить воно помилок. Метою тестування програмного забезпечення є виявлення помилок, несправностей або відсутніх вимог на відміну від фактичних вимог. Тестування програмного забезпечення можна розділити на два етапи:

- **Верифікація:** Це набір завдань, які гарантують, що програмне забезпечення правильно реалізує певну функцію. Це означає: "Чи правильно ми будуємо продукт?";
- **Валідація:** Відноситься до іншого набору завдань, які гарантують, що створене програмне забезпечення простежується відповідно до вимог замовника. Це означає "Чи ми створюємо правильний продукт?".

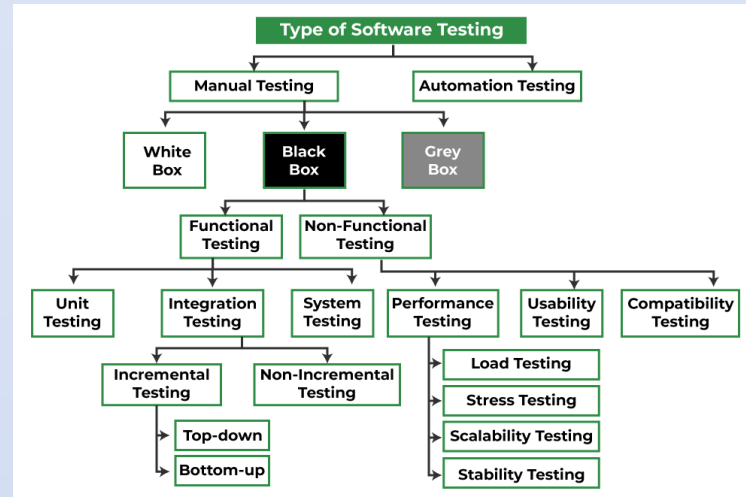
Важливість тестування програмного забезпечення:

- Дефекти можна виявити на ранній стадії: Тестування програмного забезпечення є важливим, тому що якщо є якісь помилки, їх можна виявити на ранній стадії і виправити до того, як програмне забезпечення буде готове до використання;
- Покращує якість програмного забезпечення: Тестування програмного забезпечення виявляє дефекти в програмному забезпеченні, а їх виправлення покращує якість програмного забезпечення;
- Підвищення задоволеності клієнтів: Тестування програмного забезпечення забезпечує надійність, безпеку і високу продуктивність, що призводить до економії часу, витрат і задоволеності клієнтів;

Введення в тестування ПЗ

- **Допомагає з масштабуванням:** Нефункціональне тестування програмного забезпечення допомагає виявити проблеми масштабованості та точки, в яких додаток може перестати працювати;
- **Заощаджує час і гроші:** Після запуску програми буде дуже складно відстежити і вирішити проблеми, оскільки виконання цієї роботи потребуватиме більше витрат і часу. Таким чином, краще проводити тестування програмного забезпечення через регулярні проміжки часу під час розробки програмного забезпечення.

Типи тестування ПЗ



Тестування програмного забезпечення можна умовно поділити на 3 типи:

- **Функціональне тестування (Functional testing)** - це тип тестування програмного забезпечення, який перевіряє програмні системи на відповідність функціональним вимогам. Воно виконується для того, щоб перевірити, чи працює додаток відповідно до функціональних вимог програмного забезпечення чи ні. Різновидами функціонального тестування є модульне тестування, інтеграційне тестування, системне тестування, димове тестування тощо.

Типи тестування ПЗ

- **Нефункціональне тестування (Non-functional testing)** - це тип тестування програмного забезпечення, який перевіряє додаток на наявність нефункціональних вимог, таких як продуктивність, масштабованість, портативність, навантаження тощо. Різновидами нефункціонального тестування є тестування продуктивності, стрес-тестування, юзабіліті-тестування тощо;
- **Тестування обслуговування (Maintenance testing)** - це процес зміни, модифікації та оновлення програмного забезпечення для того, щоб воно відповідало потребам замовника. Воно включає в себе регресійне тестування, яке перевіряє, що останні зміни в коді не вплинули негативно на інші раніше працюючі частини програмного забезпечення.

Типи тестування ПЗ

Окрім наведеної вище класифікації, тестування програмного забезпечення можна розділити ще на 2 способи тестування:

- **Ручне тестування** – включає в себе тестування програмного забезпечення вручну, тобто без використання будь-яких засобів автоматизації або скриптів. У цьому типі тестувальник бере на себе роль кінцевого користувача і тестує програмне забезпечення, щоб виявити будь-яку несподівану поведінку або помилку. Існують різні етапи ручного тестування, такі як модульне тестування, інтеграційне тестування, системне тестування та тестування прийнятності для користувача. Тестувальники використовують тестові плани, тестові кейси або тестові сценарії для тестування програмного забезпечення, щоб забезпечити повноту тестування. Ручне тестування також включає в себе дослідницьке тестування, коли тестувальники досліджують програмне забезпечення, щоб виявити помилки в ньому.
- **Автоматизоване тестування, яке також відоме як автоматизація тестування** - це коли тестувальник пише скрипти і використовує інше програмне забезпечення для тестування продукту. Цей процес передбачає автоматизацію ручного процесу. Автоматизоване тестування використовується для швидкого і багаторазового повторного запуску тестових сценаріїв, які були виконані вручну при ручному тестуванні. Окрім регресійного тестування, автоматизоване тестування також використовується для тестування програми з точки зору навантаження, продуктивності та стресу. Це збільшує покриття тесту, підвищує точність і економить час і гроші порівняно з ручним тестуванням.

Рівні тестування ПЗ

Тестування на рівні програмного забезпечення можна умовно поділити на 4 рівні:

- **Модульне тестування (Unit testing)** - це рівень процесу тестування програмного забезпечення, на якому тестуються окремі модулі/компоненти програмного забезпечення/системи. Мета полягає в тому, щоб підтвердити, що кожна одиниця програмного забезпечення працює так, як було задумано;
- **Інтеграційне тестування (Integration testing)** - це рівень процесу тестування програмного забезпечення, на якому окремі модулі об'єднуються і тестуються як група. Мета цього рівня тестування - виявити несправності у взаємодії між інтегрованими модулями;
- **Системне тестування (System testing)** - це рівень процесу тестування програмного забезпечення, на якому тестується повна, інтегрована система/програмне забезпечення. Мета цього тесту - оцінити відповідність системи заданим вимогам;
- **Приймальне тестування (Acceptance testing)** - це рівень процесу тестування програмного забезпечення, на якому система тестується на прийнятність. Мета цього тесту - оцінити відповідність системи бізнес-вимогам і визначити, чи є вона прийнятною для поставки.

Зупинимось з вами на модульному тестуванні!

Модульне тестування. Фреймворк JUnit

Модульне тестування передбачає тестування кожного компонента вашого коду, щоб перевірити, чи працює він так, як очікувалося. Воно ізолює кожен окремий метод вашого коду і виконує над ним тести. Юніт-тести допомагають переконатися, що ваше програмне забезпечення працює належним чином, перш ніж випустити його. Як розробник, ви будете писати модульні тести, як тільки закінчите писати частину коду.

Ви можете запитати, а хіба це не робота тестувальника? У певному сенсі так, тестувальник відповідає за тестування програмного забезпечення. Але перевірка кожного рядка коду створює значний тиск на тестувальника.

Модульне тестування в Java можна реалізувати за допомогою Junit

JUnit – фреймворк для автоматичного юніт-тестування додатків. Він містить спеціальні функції та правила, які дають змогу легко писати та запускати тести, тобто перевіряти, що кожен блок коду, або модуль, відповідальний за певну функцію програми, працює як треба.

Остання версія фреймворка - JUnit 5. Вона складається з трьох модулів: **JUnit Platform, JUnit Jupiter і JUnit Vintage**.

- **JUnit Platform** - основний модуль для управління тестами;
- **JUnit Jupiter** - модуль, який використовує нові можливості Java 8. Він надає API на основі анотацій і дає змогу працювати з модульними та динамічними тестами;
- **JUnit Vintage** - модуль для підтримки тестів, написаних з використанням JUnit 3 і JUnit 4.

Модульне тестування. Фреймворк JUnit

JUnit зручний тим, що розробник може гнучко вказувати умови тестування. Наприклад, об'єднувати тести в групи, розподіляючи їх за функціональністю, модулями, що тестуються, або рівнем критичності, прописувати умови запуску для кожного блоку коду й аналізувати результати окремо. Усе це полегшує роботу програміста або QA-інженера.

Анотації в JUnit

Анотації в JUnit - це спеціальні мітки, які Java-розробник розміщує перед методами в тестовому класі. Вони дають змогу налаштовувати процес тестування, вказуючи фреймворку, як саме їх слід обробляти. Наприклад, можна явно вказати, які з методів є тестовими випадками, які з них виконувати перед тестами і після тощо. Нижче приведено базові анотації:

- **@Test**. Цю анотацію ставимо перед методами, які належать до тестових випадків. **JUnit** зрозуміє, що їх слід виконувати як тест, а по завершенні перевірити результат;
- **@Before**. Використовується для методів, які мають бути виконані перед кожним тестовим випадком. Наприклад, якщо у нас є кілька тестів, які вимагають одних і тих самих початкових умов, ми можемо позначити метод з анотацією **@Before**, задавши необхідні умови тестування один раз;
- **@After**. Цю анотацію використовуємо перед методом, який має бути виконаний після тестового випадку;
- **@BeforeClass**, **@AfterClass**. Методи з анотацією **@BeforeClass** виконуються перед запуском першого тесту в класі, а методи з анотацією **@AfterClass** - після завершення всіх тестів у класі;
- **@Ignore**. Використовується перед методом, щоб відключити його виконання в тесті. Це може бути корисно, якщо ми не впевнені в працездатності окремих тестів і не хочемо їх використовувати, але повинні залишити в коді.
- **@BeforeEach** і **@AfterEach**. Аналоги **@Before** і **@After** в JUnit 4.

Анотації в JUnit

Повний список анотацій з докладними поясненнями та прикладами використання можна прочитати в документації:

<https://junit.org/junit5/docs/current/user-guide/#writing-tests-annotations>

Тепер розглянемо приклади анотацій в коді:

```
class TestExample{
    @BeforeEach
    public void setUp() {
        // Метод, що виконується перед кожним тестовим випадком
    }

    @AfterEach
    public void tearDown() {
        // Метод, що виконується після кожного тестового випадку
    }

    @Test
    public void testSomething() {
        // Тестовий випадок
    }

    @Test
    public void testAnotherThing() {
        // Інший тестовий випадок
    }
}
```

Анотації в JUnit

JUnit 4 та JUnit 5

JUnit 4 і JUnit 5 - це дві останні версії фреймворка JUnit, які зараз поширені в розробці. Між собою вони різняться функціональністю, синтаксисом і можливостями.

JUnit 4

Тестові методи в JUnit 4 позначаються анотацією **@Test**. Для визначення методів, що виконуються до і після тестового випадку, використовують анотації **@Before** і **@After**.

У цій версії фреймворка підтримуються параметризовані тести з використанням анотації **@RunWith(Parameterized.class)**. Це означає, що в тест можна передати параметри, необхідні для тестування.

Тести, написані на **JUnit 4**, можуть виконуватися в **JUnit 5** з використанням JUnit Vintage.

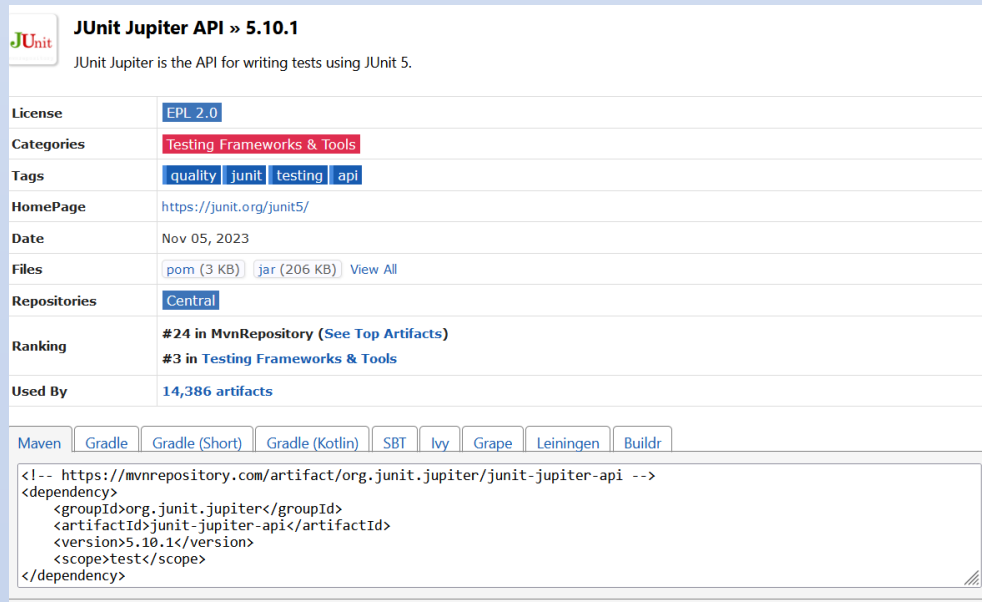
Анотації в JUnit

JUnit 5

У п'ятій версії фреймворка з'явилися нові модулі: **Jupiter** - для тестування з використанням можливостей Java 8, і **Platform** - модуль для запуску тестів. Також у JUnit 5 з'явилася можливість писати власні розширення для тестів і запускати їх за анотацією **@ExtendWith**. За аналогією з JUnit 4 підтримуються параметризовані тести з анотацією **@ParameterizedTest**.

У прикладах із кодом ми використовуватимемо JUnit 5 - це сучасна версія фреймворку, яка підтримує Java 8 і JUnit 4.

Для встановлення **JUnit** до вашого проекту в **IntelliJ IDEA** використаємо **Maven**. Отримаємо залежність для ***pom.xml*** файлу за посиланням <https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-api/5.10.1>



The screenshot shows the Maven Central repository page for the artifact **JUnit Jupiter API » 5.10.1**. The page includes the JUnit logo and a description: "JUnit Jupiter is the API for writing tests using JUnit 5." Below this, there is a table with the following information:

License	EPL 2.0
Categories	Testing Frameworks & Tools
Tags	quality junit testing api
HomePage	https://junit.org/junit5/
Date	Nov 05, 2023
Files	pom (3 KB) jar (206 KB) View All
Repositories	Central
Ranking	#24 in MvnRepository (See Top Artifacts) #3 in Testing Frameworks & Tools
Used By	14,386 artifacts

Below the table, there are tabs for different build systems: Maven, Gradle, Gradle (Short), Gradle (Kotlin), SBT, Ivy, Grape, Leiningen, and Buildr. The Maven tab is selected, and it shows the following XML snippet for the dependency:

```
<!-- https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-api -->
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-api</artifactId>
  <version>5.10.1</version>
  <scope>test</scope>
</dependency>
```

Приклад роботи JUnit

Напишемо на Java простий калькулятор:

```
class Calculator {  
    no usages  
    public int add(int a, int b) {  
        return a + b;  
    }  
  
    no usages  
    public int subtract(int a, int b) {  
        return a - b;  
    }  
  
    no usages  
    public int multiply(int a, int b) {  
        return a * b;  
    }  
  
    no usages  
    public int divide(int a, int b) {  
        if (b == 0)  
            throw new IllegalArgumentException("Cannot divide by zero");  
        return a / b;  
    }  
}
```

Для модульного тестування калькулятора нам потрібно написати окремі тести для додавання, віднімання, множення і два тести для ділення. З **JUnit** код буде такий:

Приклад роботи JUnit

```
class CalculatorTest {

    @Test
    public void testAddition() {
        Calculator calculator = new Calculator();
        int result = calculator.add( a: 3, b: 5);
        Assertions.assertEquals( expected: 8, result);
    }

    @Test
    public void testSubtraction() {
        Calculator calculator = new Calculator();
        int result = calculator.subtract( a: 10, b: 4);
        assertEquals( expected: 6, result);
    }

    @Test
    public void testMultiplication() {
        Calculator calculator = new Calculator();
        int result = calculator.multiply( a: 6, b: 3);
        assertEquals( expected: 18, result);
    }

    @Test
    public void testDivision() {
        Calculator calculator = new Calculator();
        int result = calculator.divide( a: 10, b: 2);
        assertEquals( expected: 5, result);
    }

    @Test
    public void testDivisionByZero() {
        Calculator calculator = new Calculator();
        assertThrows(IllegalArgumentException.class, () -> calculator.divide( a: 10, b: 0));
    }
}
```

Розберемо його, тут ми імпортували анотацію Test із фреймворку **JUnit**. Вона позначає методи як тестові випадки, визначаючи їх виконання під час запуску тестів. Також ми імпортували статичні методи тверджень (assertions) з класу **Assert** - **assertEquals(expected, actual)**.

Приклад роботи JUnit

Вони порівнюють очікувані та фактичні результати тестів. Якщо результати не збігаються, то тест вважається не пройденим.

public class CalculatorTest {... } - визначили клас для наших тестів. Далі ми прописали тестові методи, наприклад **testAddition()**, **testSubtraction()**, **testMultiplication()**, **public void testDivision()**. Усередині кожного методу тестуємо конкретну арифметичну операцію. Для цього ми порівнюємо результат роботи калькулятора із заздалегідь підбраною правильною відповіддю за допомогою **assertEquals**. Для кожного тесту створили екземпляр класу **Calculator**, який буде використовуватися для їх проведення. У цьому прикладі ми спочатку написали програму, а потім - тести для неї. Але іноді розробники використовують інший підхід.

Test-driven development

Test-driven development (TDD) - це підхід до розроблення програм, за якого розробник спочатку описує тести для функції, яку хоче створити, а потім пише сам код, який проходить ці тести. За такого підходу головні витрати розробки - час на рефакторинг і виправлення помилок - знижуються. А отже, зменшуються і витрати на створення та підтримку продукту.

Спрощено TDD можна представити у вигляді кількох кроків:

- **Написання тесту.** Розробник починає роботу зі створення тесту, який перевірятиме працездатність коду;
- **Запуск тесту.** Під час першого запуску тест не повинен бути пройдений, оскільки функціонального коду програми ще немає;
- **Написання коду.** Розробник пише код із мінімальною функціональністю, яка дає змогу успішно пройти тест;
- **Повторний запуск тесту.** При повторному запуску тест має бути пройдений вдало;
- **Рефакторинг.** Після успішного завершення тестів розробник може приступити до рефакторингу - поліпшення та оптимізації коду. Важливо, що після кожної зміни запуск тесту повторюється.

Кожен функціональний модуль, який додається до додатка, має пройти ці етапи. Таким чином, розробник уже в процесі написання окремих частин програми підтверджує, що вони успішно проходять тести.

Для прикладу створимо програму для обчислення факторіалу числа. Спочатку напишемо тести, а потім функціональний код.

Test-driven development

```
class NumberUtilTest {  
    @Test  
    public void testFactorial() {  
        NumberUtil util = new NumberUtil();  
        int result = util.factorial(5); // Визначаємо факторіал числа 5  
        assertEquals( expected: 120, result); // Порівнюємо результат із правильною відповіддю  
    }  
}
```

```
class NumberUtil {  
    2 usages  
    public int factorial(int n) {  
        if (n < 0) {  
            throw new IllegalArgumentException("Факторіал не може бути розрахований для від'ємних чисел");  
        }  
        if (n == 0 || n == 1) {  
            return 1;  
        }  
        return n * factorial(n - 1);  
    }  
}
```

Запустимо тест, якщо все було виконано правильно, то отримаємо таке виведення в консоль:

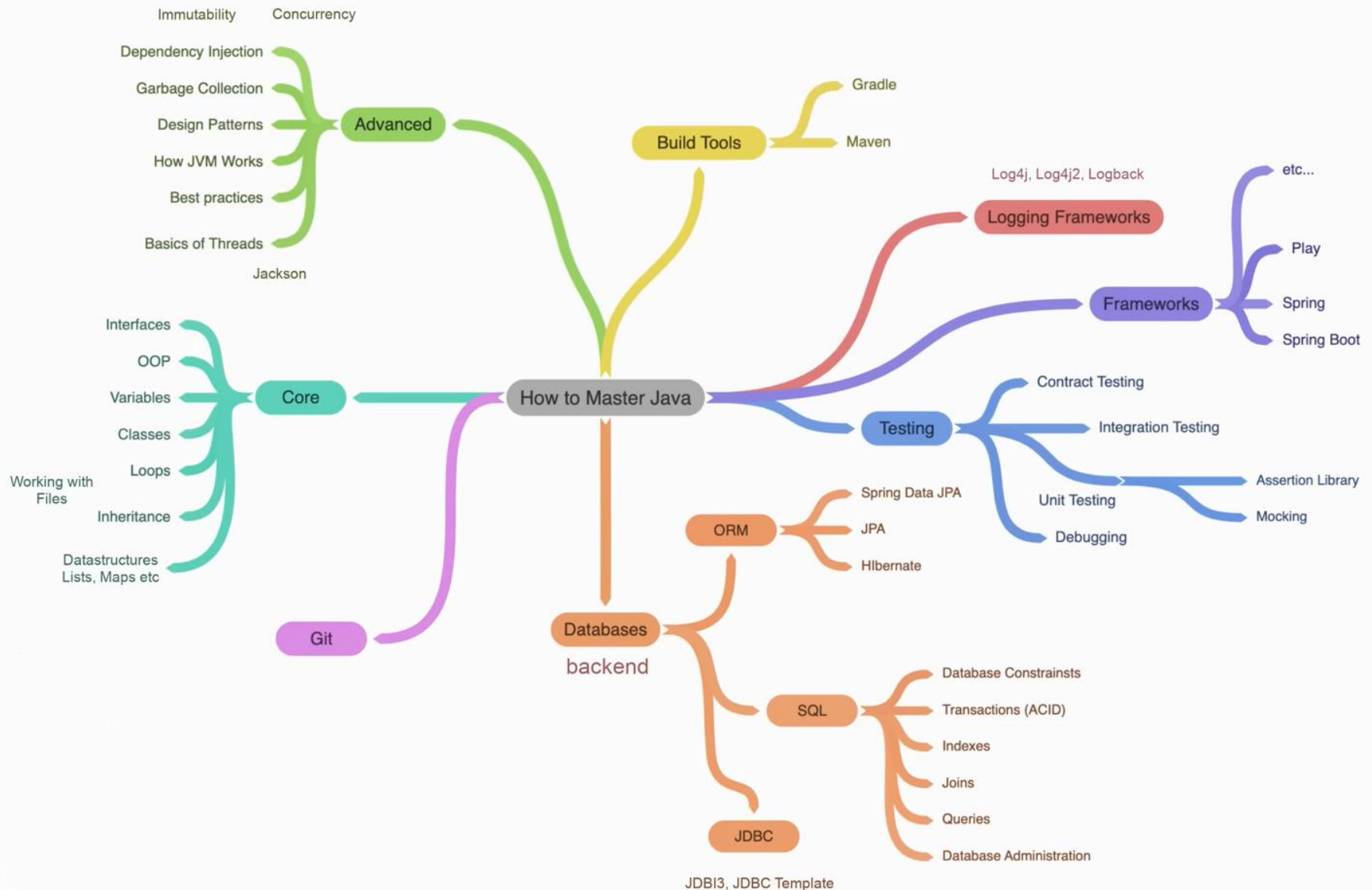
```
✓ NumberUtilTest (ua.aharoo) 18 ms  
✓ testFactorial() 18 ms  
✓ Tests passed: 1 of 1 test – 18 ms  
"C:\Program Files\Java\jdk-20\bin\java.exe" ...  
Process finished with exit code 0
```

Контрольні питання

1. Охарактеризуйте термін "тестування програмного забезпечення" у відповідності до сфери інформаційних технологій.
2. Поясніть важливість тестування програмного забезпечення та його роль у розробці програм.
3. Які існують рівні тестування програмного забезпечення, і як вони взаємодіють у загальній стратегії тестування?
4. Навіщо використовується фреймворк JUnit у тестуванні програмного забезпечення?
5. Охарактеризуйте анотації в JUnit та їх роль у структурі тестових сценаріїв.
6. Охарактеризуйте концепцію test-driven development (TDD) та поясніть, як вона взаємодіє з процесом розробки програмного забезпечення.

What`s next? (Що далі?)

Шлях розвитку Java-розробника



The END
Lec 14.

Рекомендована література

1. Naughton P., Schildt H. Java 2: The Complete Reference, Third Edition 3rd Edition / P. Naughton, H. Schildt – 3rd ed. – Mcgraw-Hill Osborne Media, 1999. – 1008 p.
2. Mak G. Spring Enterprise Recipes: A Problem-Solution Approach / G. Mak, J. Long; 2nd ed. – NY. : Springer, 2009. – 1104 p.
3. Mike Keith, ol. Pro JPA 2. Mastering the Java™ Persistence API / Mike Keith. Merrick Schincari. – New York : Apress, 2009. – 238 p.
4. Calvert K. L. TCP/IP Sockets in Java Practical Guide for Programmers / K. L. Calvert, M. J. Donahoo. – 2nd ed. – Burlington : Morgan Kaufmann, 2007. – 193 p.
5. 11. Heffelfinger D. Java EE 6 with GlassFish 3 Application Server / D. Heffelfinger – Packt Publishing.